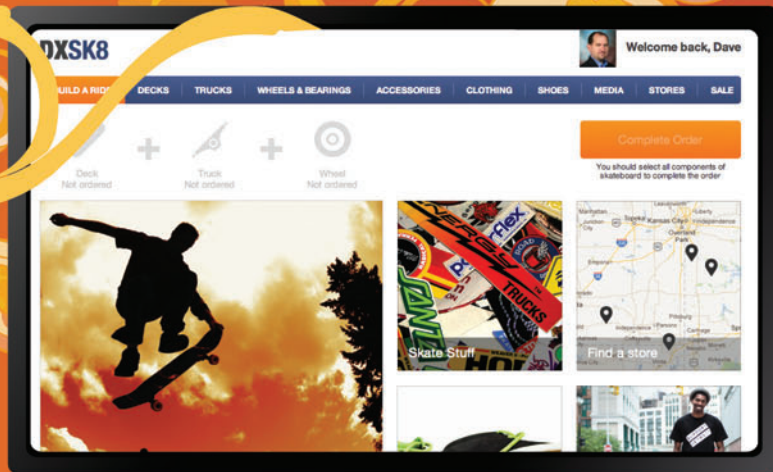




Imagine. Create. Deploy.

Inspired? So Are We.



Inspiration is all around us. From beautiful screens on the web to well-designed reports. New devices push the development envelope and asks that we consider new technologies. The latest release, DevExpress 12.2, delivers the tools you need to build the multi-channel solutions you can imagine: Windows 8-inspired applications with live tiles perfect for Microsoft Surface, multi-screen iOS and Android apps. It's all possible. Let's see what develops.



Download your 30-day trial
at www.DevExpress.com

DXv2

The next generation of inspiring tools. **Today.**



Copyright 1998 - 2012 Developer Express Inc. All rights reserved. All trademarks are property of their respective owners.

msdn magazine



Access Online Services with
the Windows Runtime..... 36

Access Online Services with the Windows Runtime and OAuth Tim Kulp	36
TypeScript: Making .NET Developers Comfortable with JavaScript Shayne Boyer	46
The C# Memory Model in Theory and Practice, Part 2 Igor Ostrovsky	52
Building Hypermedia Web APIs with ASP.NET Web API Pablo Cibraro	58
Version Control in the TFS Client Object Model Jeff Bramwell	64

COLUMNS

CUTTING EDGE

Essential Facebook Programming:
Building a Windows Client
Dino Esposito, page 6

WINDOWS WITH C++

The Evolution of Threads
and I/O in Windows
Kenny Kerr, page 12

DATA POINTS

Shrink EF Models with
DDD Bounded Contexts
Julie Lerman, page 22

WINDOWS AZURE INSIDER

Windows Azure Web Sites:
Quick-and-Easy Hosting as a Service
Bruno Terkaly and
Ricardo Villalobos, page 28

TEST RUN

Artificial Immune Systems for
Intrusion Detection
James McCaffrey, page 68

THE WORKING PROGRAMMER

.NET Collections:
Getting Started with C5
Ted Neward, page 74

DIRECTX FACTOR

Windows 8 Sound Generation
with XAudio2
Charles Petzold, page 76

DON'T GET ME STARTED

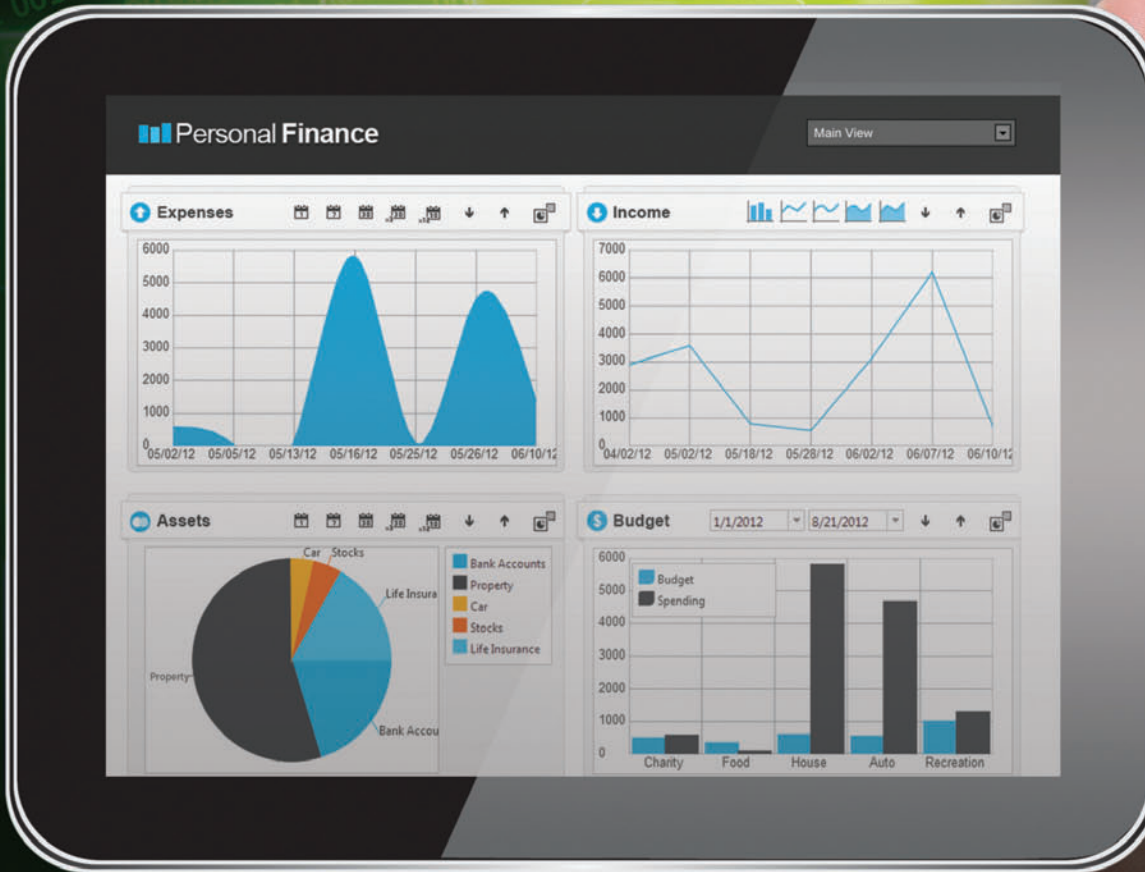
Lowering Higher Education, Again
David Platt, page 80

Compatible with
Microsoft® Visual Studio® 2012

San Francisco

Drop Filter Fields here

Country	Population	GDP Per Capita	Life Expectancy
Portugal	11	11433	79
Romania	23	2845	73
Serbia	11	1810	74
Slovak Republic	6	8591	75
Slovenia	3	13784	78
Spain	43	16306	81
Sweden	19	32338	83
Switzerland	8	37872	82
Ukraine	47	1136	68
United Kingdom	61	28955	79
Yemen	1	17966	57



BRILLIANT UX

At Your Fingertips



Home

Add Customer Delete Customer Export Customer Visualizations

Review Data Plotting Customers Sales By Category Only Sales

Drop Filter Fields here

Estimate

Product Name	Country	All Periods
France		1002.7500
Germany		2042.2300
Spain		479.7900
Venezuela		663.6200
Austria		1169.2800

Dosage	Unit	Frequen
650	mg.	PO PRN
375	mg.	PO PRN
250	mg.	PO Q12H
250	mg.	PO PRN
250	mg.	PO PRN

Healthcare Dashboard Tue Aug 21 2012

Patient Admissions

Name	Severity
AmCity Clinic	
JELDREE Duncan	
RIZZO John	
RIZZO John	

Vital Signs

Time
6/25/2009
6/25/2009
6/25/2009
6/25/2009



Vital Signs

Vital Sign: Blood Pressure (B)
5/2009
5/2009
5/2009
5/2009

Value

140
125
145
130

Download your free trial
infragistics.com/EXPERIENCE

 **INFRAGISTICS™**
DESIGN / DEVELOP / EXPERIENCE

Infragistics Sales US 800 231 8588 • Europe +44 (0) 800 298 9055 • India +91 80 4151 8042 • APAC +61 3 9982 4545

Copyright 1996-2013 Infragistics, Inc. All rights reserved. Infragistics and NetAdvantage are registered trademarks of Infragistics, Inc. The Infragistics logo is a trademark of Infragistics, Inc. All other trademarks or registered trademarks are the respective property of their owners.



dtSearch®

Instantly Search Terabytes of Text

- 25+ fielded and full-text search types
- dtSearch's **own document filters** support "Office," PDF, HTML, XML, ZIP, emails (with nested attachments), and many other file types
- Supports databases as well as static and dynamic websites
- Highlights hits in all of the above
- APIs for .NET, Java, C++, SQL, etc.
- 64-bit and 32-bit; Win and Linux

"lightning fast" Redmond Magazine

"covers all data sources" eWeek

"results in less than a second" InfoWorld

hundreds more reviews and developer case studies at www.dtsearch.com

dtSearch products:

- ◆ Desktop with Spider
- ◆ Web with Spider
- ◆ Network with Spider
- ◆ Engine for Win & .NET
- ◆ Publish (portable media)
- ◆ Engine for Linux
- ◆ Document filters also available for separate licensing

Ask about fully-functional evaluations

The Smart Choice for Text Retrieval® since 1991

www.dtSearch.com 1-800-IT-FINDS

msdn

magazine

JANUARY 2013 VOLUME 28 NUMBER 1

BJÖRN RETTIG Director

MOHAMMAD AL-SABT Editorial Director/mmeditor@microsoft.com

PATRICK O'NEILL Site Manager

MICHAEL DESMOND Editor in Chief/mmeditor@microsoft.com

DAVID RAMEL Technical Editor

SHARON TERDEMAN Features Editor

WENDY HERNANDEZ Group Managing Editor

KATRINA CARRASCO Associate Managing Editor

SCOTT SHULTZ Creative Director

JOSHUA GOULD Art Director

SENIOR CONTRIBUTING EDITOR Dr. James McCaffrey

CONTRIBUTING EDITORS Rachel Appel, Dino Esposito, Kenny Kerr, Julie Lerman, Ted Neward, Charles Petzold, David S. Platt, Bruno Terkaly, Ricardo Villalobos

Redmond Media Group

Henry Allain President, Redmond Media Group

Doug Barney Vice President, New Content Initiatives

Michele Imgrund Sr. Director of Marketing & Audience Engagement

Tracy Cook Director of Online Marketing

ADVERTISING SALES: 508-532-1418/mmorollo@1105media.com

Matt Morollo VP/Group Publisher

Chris Kourtoglou Regional Sales Manager

William Smith National Accounts Director

Danna Vedder National Account Manager/Microsoft Account Manager

Jenny Hernandez-Asandas Director, Print Production

Serena Barnes Production Coordinator/msdnadproduction@1105media.com

1105 MEDIA

Neal Vitale President & Chief Executive Officer

Richard Vitale Senior Vice President & Chief Financial Officer

Michael J. Valenti Executive Vice President

Christopher M. Coates Vice President, Finance & Administration

Erik A. Lindgren Vice President, Information Technology & Application Development

David F. Myers Vice President, Event Operations

Jeffrey S. Klein Chairman of the Board

MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. \$35.00, International \$60.00. Annual digital subscription rates payable in U.S. funds are: U.S. \$25.00, International \$25.00. Single copies/back issues: U.S. \$10, all others \$12. Send orders with payment to: MSDN Magazine, PO Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. POSTMASTER: Send address changes to MSDN Magazine, PO Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: PO Box 201, Richmond Hill, ON L4B 4R5, Canada.

Printed in the U.S.A. Reproductions in whole or part prohibited except by written permission. Mail requests to "Permissions Editor," c/o MSDN Magazine, 4 Venture, Suite 150, Irvine, CA 92618.

Legal Disclaimer: The information in this magazine has not undergone any formal testing by 1105 Media, Inc. and is distributed without any warranty expressed or implied. Implementation or use of any information contained herein is the reader's sole responsibility. While the information has been reviewed for accuracy, there is no guarantee that the same or similar results may be achieved in all environments. Technical inaccuracies may result from printing errors and/or new developments in the industry.

Corporate Address: 1105 Media, Inc., 9201 Oakdale Ave., Ste 101, Chatsworth, CA 91311, www.1105media.com

Media Kits: Direct your Media Kit requests to Matt Morollo, VP Publishing, 508-532-1418 (phone), 508-875-6622 (fax), mmorollo@1105media.com

Reprints: For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International, Phone: 212-221-9595, E-mail: 1105reprints@parsintl.com, www.magreprints.com/QuickQuote.asp

List Rental: This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Merit Direct. Phone: 914-368-1000; E-mail: 1105media@meritdirect.com; Web: www.meritdirect.com/1105

All customer service inquiries should be sent to MSDNmag@1105service.com or call 847-763-9560.



Printed in the USA



The world's leading Imaging SDK NOW **RUNS EVERYWHERE**

OCR

BARCODE

PDF & PDF/A

FORMS RECOGNITION &
PROCESSING

DOCUMENT
PREPROCESSING

ANNOTATIONS

150 +
FILE FORMATS

SCANNING

DICOM & PACS

MEDICAL
WORKSTATION

IMAGE PROCESSING

VIRTUAL PRINTER

MPEG-2 TRANSPORT
STREAM

MULTIMEDIA PLAYBACK &
CAPTURE

DVD & DVR

CODECS

COMPREHENSIVE IMAGING SDK FOR WIN 32/64 WinRT, MAC, iOS, ANDROID & LINUX





The X Factor

If you've been reading *MSDN Magazine* long enough—and we've only been around in one form or another for a quarter-century now—you know that columns come and columns go, but timeless programming challenges remain.

Look no further than this month's launch of the new DirectX Factor column. Charles Petzold is setting aside his Touch and Go column, with its focus on managed, Windows Phone mobile and touch UI development, to explore the new arena of native C++ development for the DirectX API in the Windows Runtime.

Of course, Petzold is a guy who's not afraid to jump on a bandwagon early. The author of the book "Programming Windows" (Microsoft Press, 1988), now in its sixth edition, actually wrote the first-ever article in *MSDN Magazine* on Windows programming, way back in December 1986. Since then, he's worked tirelessly to explore and explain new development platforms and technologies in the pages of *MSDN Magazine*. The arrival of Windows 8 and the Windows Runtime—with its support for both managed and native development—certainly gives Petzold plenty to explore and explain.

"As Windows runs on smaller and less-hefty processors on tablets and other mobile devices, and as we strive to make the UIs of our Windows applications become faster and more fluid, C++ and DirectX have become more important than ever. This is what I'd like to explore in this column," Petzold says.

Make no mistake: Managed languages like C# and Visual Basic aren't going anywhere, and robust platforms such as Windows Phone continue to advance and evolve. Petzold says he expects to see more solutions where applications are coded in multiple languages, with developers challenged to intelligently choose which languages should be used where.

"Programmers will identify and isolate the time-critical and graphics-intensive stuff, and code that in C++ in the form of a Windows Runtime Component library, which can then be accessed by C# or JavaScript code," he explains.

Petzold says that all his sample programs will reflect this approach, as he combines DirectX with XAML-based UIs. "That's something we really haven't been able to do before, and I'm thrilled to have the opportunity to write about these techniques," he says.

Petzold isn't abandoning his mobile focus, either. The Windows Phone 8 API supports native C++ code and DirectX, and Petzold plans to explore how developers can write DirectX code that can run on both Windows 8 and Windows Phone 8 hardware. And as was the case with Touch and Go, Petzold plans to continue delving into issues and techniques related to touch interfaces.

"As we developers work with more intimate touch-based UIs, I'm particularly interested in how we can use touch to manipulate complex graphical objects," Petzold says. "I actually think this is one of the keys to giving rather bulky fingers more precision on the screen."

But perhaps I'm getting ahead of myself. In this issue's column, which begins on p. 76, Petzold looks into the XAudio2 sound-generation component in Windows 8, and he plans to move on to exploring 3D graphics within the next few months. Give the new column a read and let us know your opinion. What would you like to see Petzold cover in the DirectX Factor column? E-mail me at mmeditor@microsoft.com.

Lowering Higher Education, Again

I can't close without making brief mention of this month's Don't Get Me Started column (p. 80). David Platt returns to a topic he covered a year ago in our pages—the coming, Internet-driven revolution in higher education. As Platt notes, colleges and universities are poised to experience many of the same dislocations that have challenged the newspaper and music industries.

And for good reason. In his January 2012 column, David explained that the inflation-adjusted cost of a college education has *quadrupled* since 1982. As a guy with three kids (including two who will enter college in the next three years), the spiraling cost of higher education has me looking hard at alternatives.

Could emerging, massive open online course (MOOC) providers offer a cost-effective alternative to four-year colleges? And how could MOOCs such as Udacity, edX and Coursera impact the education and ongoing training of software developers? Read Platt's thought-provoking column and let us know what you think.

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

© 2013 Microsoft Corporation. All rights reserved.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, you are not permitted to reproduce, store, or introduce into a retrieval system *MSDN Magazine* or any part of *MSDN Magazine*. If you have purchased or have otherwise properly acquired a copy of *MSDN Magazine* in paper format, you are permitted to physically transfer this paper copy in unmodified form. Otherwise, you are not permitted to transmit copies of *MSDN Magazine* (or any part of *MSDN Magazine*) in any form or by any means without the express written permission of Microsoft Corporation.

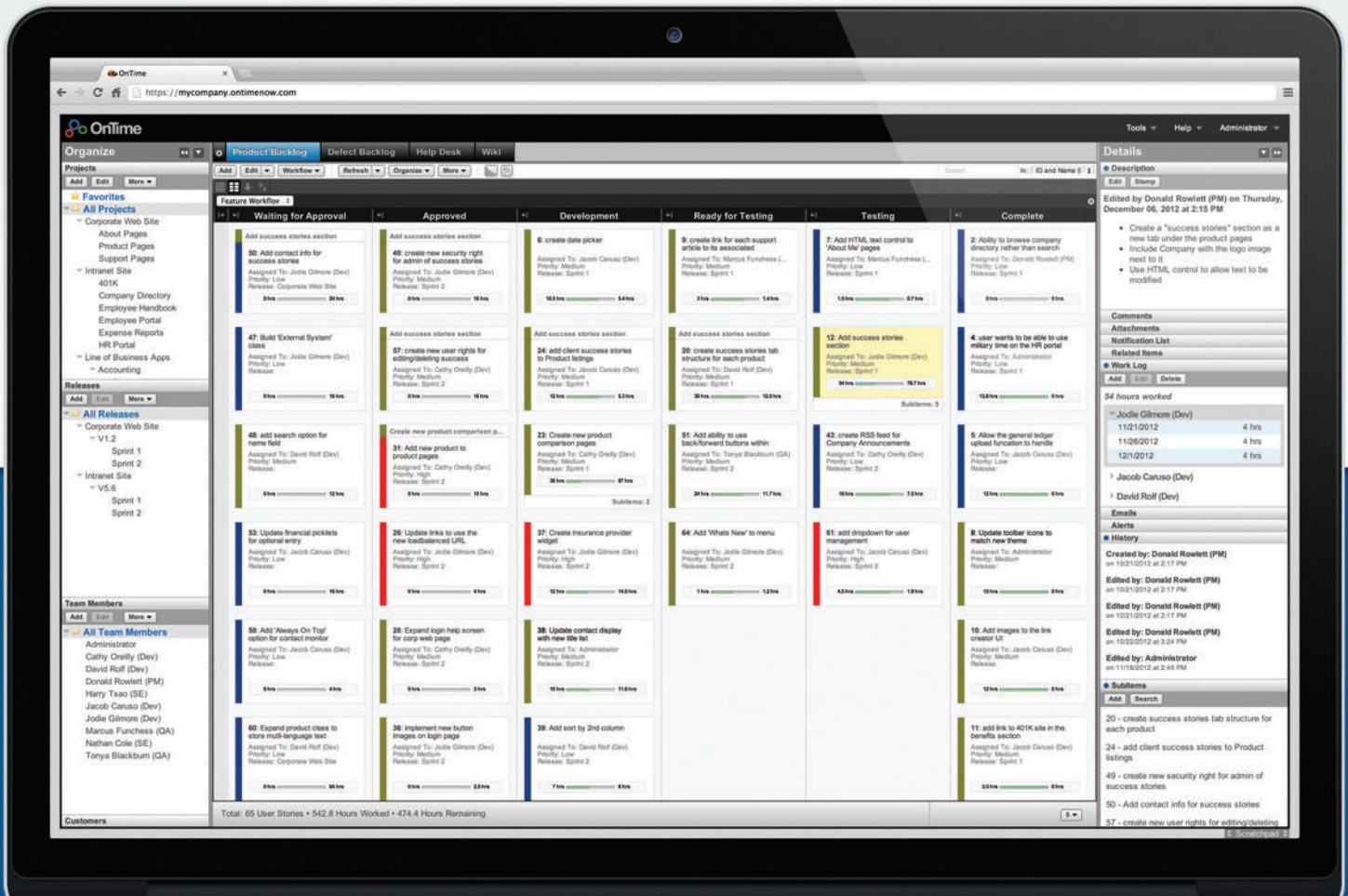
A listing of Microsoft Corporation trademarks can be found at microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx. Other trademarks or trade names mentioned herein are the property of their respective owners.

MSDN Magazine is published by 1105 Media, Inc. 1105 Media, Inc. is an independent company not affiliated with Microsoft Corporation. Microsoft Corporation is solely responsible for the editorial contents of this magazine. The recommendations and technical guidelines in *MSDN Magazine* are based on specific environments and configurations. These recommendations or guidelines may not apply to dissimilar configurations. Microsoft Corporation does not make any representation or warranty, express or implied, with respect to any code or other information herein and disclaims any liability whatsoever for any use of such code or other information. *MSDN Magazine*, MSDN, and Microsoft logos are used by 1105 Media, Inc. under license from owner.



OnTime Scrum

Agile project management
& bug tracking software



Introducing OnTime 13.

Ditch your sticky notes. The Card View is now here.

The **OnTime Card View** is the ideal planning board tool for Kanban or Scrum teams. It adds a whole new dimension to user story management, bug tracking and workflow automation.

Learn more about Card View and the many other features of OnTime Scrum that your dev team will love.

OnTimeNow.com/MSDN

\$834 per month
for up to 10 users
billed annually

special small-team pricing

\$584 per user
per month
billed annually

for teams of 11+ users



800.653.0024 • www.ontimenow.com • www.axosoft.com • @axosoft



Essential Facebook Programming: Building a Windows Client

In my previous column I discussed the basics of Facebook programming in the context of a Web site—specifically an ASP.NET MVC Web site (you can read the article at msdn.microsoft.com/magazine/jj863128). As you can imagine, the use of the Facebook API isn't limited to Web apps, even though one of its most common uses is just to authenticate users of a new Web site in a “softer” manner.

In this column I'll build a Windows Presentation Foundation (WPF) client app that uses the Facebook API to authenticate users and post updates with pictures. Because the Web is a different environment from the desktop, this column obviously will be different from what I wrote last month. The Facebook API, however, is the same. You can get the Facebook C# SDK via NuGet (see facebookapi.codeplex.com for more details on the Facebook API for C#.NET developers).

Integrating Apps with Social Networks

Most social networks have a similar architecture. Interaction between users and the engine of the social network happens in either of the following two ways: directly via the main Web site (for example, the Twitter or Facebook site) or through the mediation of an app hosted internally by the social network site. For the sake of clarity, I'll refer to these apps as connectors.

A client app (Web site, WPF or Windows Phone) can only interact with the engine of the social network through the mediation of a connector. For the sample code accompanying this column, I'll use a Facebook connector called Memento (note that the code, provided for example purposes, comes as a single project, not a full solution, and uses Visual Studio 2010). The name Memento—and its related icon—will show up as the footer of any wall posts made by any client app using Memento to connect to Facebook. You can create a connector for Facebook at developers.facebook.com/apps.

Connectors are characterized by a pair of unique strings—app key and secret—and, more interestingly for our purposes, can serve a variety of client apps, whether Web, mobile or desktop. **Figure 1** summarizes the overall architecture of social networks when it comes to interaction.

In the end, any user apps that need to integrate with most popular social networks (for example, Facebook, Twitter or Foursquare) are essentially built as clients of a social network-specific connector app.

For an app, integrating with a social network essentially means accessing the social network functionalities on behalf of a registered

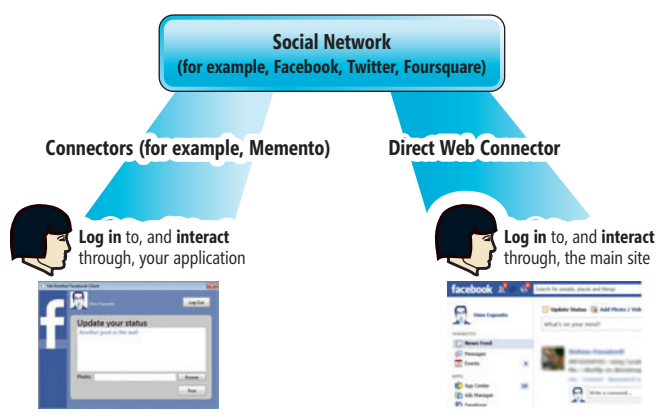


Figure 1 The Interaction Model of Social Networks

user. This involves a couple actions: authenticating users against the social network and performing the actual action, such as making a check-in for a given venue, posting on the user's wall or tweeting.

This is the essence of integrating a social network API into a custom app and, as it turns out, it might not be limited to just Web apps.

Differences from a Windows App

In last month's column I first discussed how to authenticate users of a Web site via Facebook, then I presented some code to post both interactively and programmatically to the user's wall. Posting to a user's wall doesn't require a different approach if done from a Windows app; authentication, instead, requires some adjustments.

A social network is primarily a Web app and exposes authentication services via the OAuth protocol. An app that wants to authenticate its users via Facebook accounts, for example, needs to place an HTTP call to some Facebook endpoint. The endpoint, however,

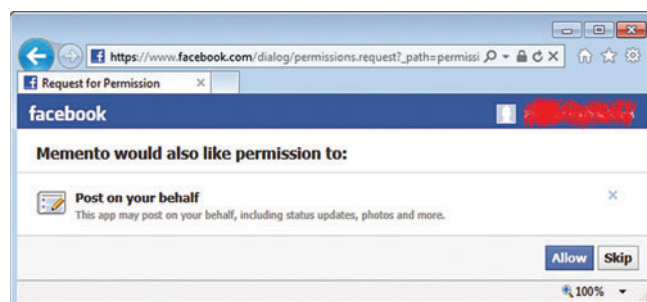


Figure 2 The Memento Connector Explicitly Asks the User for Permissions

Code download available at archive.msdn.microsoft.com/mag201301CuttingEdge.

The #1 Native Toolset for Building Windows 8 Apps

Build for the consumer and enterprise markets
with either XAML or HTML



www.telerik.com/win8

 **telerik**
deliver more than expected

Figure 3 Code to Get Login URL

```
public static String GetLoginUrl()
{
    var client = new FacebookClient();
    var fbLoginUri = client.GetLoginUrl(new
    {
        client_id = ConfigurationManager.AppSettings["fb_key"],
        redirect_uri =
            "https://www.facebook.com/connect/login_success.html",
        response_type = "code",
        display = "popup",
        scope = "email,publish_stream"
    });

    return fbLoginUri.ToString();
}
```

returns an HTML page for the actual user to interact with by entering credentials and authorizing the connector to operate on her behalf (see **Figure 2**).

The part that's different in a Windows or mobile scenario compared to a Web app is how you handle the redirect to the HTML page that the social network provides for credentials and permissions.

Yet Another Facebook Windows Client

Let's create a WPF project and add a bit of XAML markup to the main window. At the minimum, you need to have a couple of buttons to trigger the login and logout process, and a label to display the name of the currently logged-in user. In addition, you can have an image element to display the picture of the current user. What else? Let's look at the code you put in the Login click handler:

```
var loginUrl = FbHelpers.GetLoginUrl();
```

The first step is getting the login URL from Facebook. The code to get the URL is the same as you'd use in a Web scenario, as shown in **Figure 3**.

As you might notice, the `redirect_uri` parameter (required) points to a Facebook success endpoint. That's the landing page for after the login process has terminated. In a Web scenario you'd use the current page, so the user is first redirected to the Facebook site and then back to the original requesting page.

The `GetLoginUrl` method only gets you the URL to call to log in. In a Web scenario, you simply invoke a redirect to switch to the new page and display the typical Facebook UI for login or—if the user is already logged in on the computer in use—the permission page as shown in **Figure 2**. To achieve the same in a Windows app, you need an initially hidden `WebBrowser` control in the UI. Here's the full code for the click handler of the app's login button:

```
public void Login()
{
    var loginUrl = FbHelpers.GetLoginUrl();
    ShowBrowser = true;
    view.Browser.Navigate(loginUrl);
}
```

The sample app available for download uses the Model-View-ViewModel (MVVM) pattern to abstract away UI details. Therefore, setting `ShowBrowser` to true just has the effect of turning on the visibility of the `WebBrowser` control. Finally, the `Navigate` method directs the component to the specified URL. **Figure 4** shows the status of the sample app before and after the login click.

The authentication process might take up to two steps. If the user is already logged in to the social network, the Web browser directly receives the landing page. If the user is not logged in, the page shown in **Figure 4** is presented. However, if the user isn't associated with the connector being used by the client app, then an intermediate screen like that shown in **Figure 2** explicitly asks to grant permissions.

How do you handle the load of pages in the `WebBrowser` component? You basically need a `Navigated` event handler:

```
void facebookBrowser_Navigated(Object sender, NavigationEventArgs e)
{
    var fb = new FacebookClient();
    FacebookOAuthResult oAuthResult;
    if (!fb.TryParseOAuthCallbackUrl(e.Uri, out oAuthResult))
        return;

    if (oAuthResult.IsSuccess)
        _viewPresenter.LoginSucceeded(oAuthResult);
    else
        _viewPresenter.LoginFailed(oAuthResult);
}
```

Facebook indicates whether the login process completed successfully or not. Note that the login fails if the user simply denies requested permissions to the connector. If the login fails, just reset the UI hiding the Web browser and display some feedback to the user. It's much more interesting when the login is successful, as shown in **Figure 5**.

The access token is a string you get from the social network in return for a successful login code.

The code in **Figure 5** is similar to what you need to use in a Web scenario. Note that once the user has logged in successfully, the app is still not yet ready to operate on behalf of the user. Moreover, the app doesn't yet know anything about the user—not even the user's name or some sort of ID. This means that persisting authentication data isn't possible yet. Therefore, another step is mandatory for both Web and Windows scenarios: getting the access token.

The Access Token

The access token is a string you get from the social network in return for a successful login code. The login code simply indicates

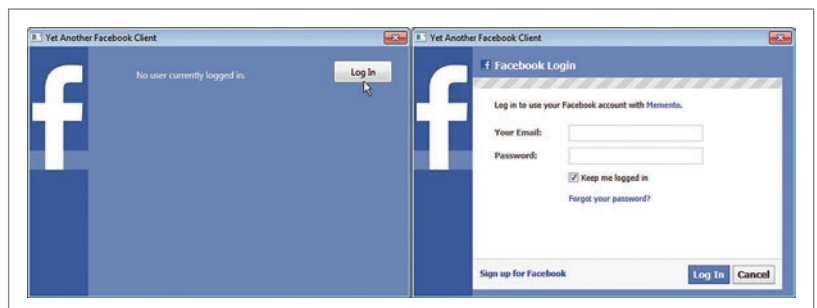


Figure 4 First Step of Login to Facebook

CONVERT PRINT CREATE MODIFY & COMBINE

FILES

Aspose.Words

DOC, DOCX, RTF, HTML, PDF,
XPS & other document formats.

Aspose.Cells

XLS, XLSX, XLSM, XLTX, CSV,
SpreadsheetML & image formats.

Aspose.Pdf

PDF, XML, XLS-FO, HTML, BMP,
JPG, PNG & other image formats.

Aspose.Email

MSG, EML, PST, EMLX &
other formats.



.NET Java SSRS JasperReports
SharePoint

Follow us on
Facebook & Twitter



Scan our QR Code
for an exclusive
20% coupon code.



Get your FREE evaluation copy at <http://www.aspose.com>

US Sales: 1.888.277.6734
sales@aspose.com

EU Sales: +44 (0) 141 416 1112
sales.europe@aspose.com

AU Sales: +61 2 8003 5926
sales.asiapacific@aspose.com

Figure 5 Upon a Successful Login

```
public void LoginSucceeded(FacebookOAuthResult oauthResult)
{
    // Hide the Web browser
    ShowBrowser = false;

    // Grab the access token (necessary for further operations)
    var token = FbHelpers.GetAccessToken(oauthResult);
    Token = token;

    // Grab user information
    dynamic user = FbHelpers.GetUser(token);

    // Update the user interface
    UserName = String.Format("{0} {1}", user.first_name,
        user.last_name);
    UserPicture = user.picture;
    IsLogged = true;
}
```

Figure 6 Code to Get Access Token After Successful OAuth Login

```
public static String GetAccessToken(FacebookOAuthResult oauthResult)
{
    var client = new FacebookClient();
    dynamic result = client.Get("/oauth/access_token",
        new
        {
            client_id = ConfigurationManager.AppSettings["fb_key"],
            client_secret =
                ConfigurationManager.AppSettings["fb_secret"],
            redirect_uri =
                "https://www.facebook.../login_success.html",
            code = oauthResult.Code
        });

    return result.access_token;
}
```

that the user who proceeded with authentication is known to the network. The access code ties together the user identity and the connector. The access code tells the social network engine whether the user app has enough permission to perform the requested operation on behalf of that user. The same user has a different access token for different connectors.

Figure 6 shows the C# code to get the access token after a successful OAuth login.

In a Web app, you might want to persist the access token in a custom cookie or as extra data in a custom authentication cookie managed by a custom *IPrincipal* object. In a Windows scenario, you might want

Figure 8 Code to Post an Update with an Attached Picture

```
public static void PostWithPhoto(
    String token, String status, String photoPath)
{
    var client = new FacebookClient(token);
    using (var stream = File.OpenRead(photoPath))
    {
        client.Post("me/photos",
            new
            {
                message = status,
                file = new FacebookMediaStream
                {
                    ContentType = "image/jpg",
                    FileName = Path.GetFileName(photoPath)
                }.SetValue(stream)
            });
    }
}
```

to resort to local storage. When the user logs out of a Windows app, you simply clear out any stored data. Once you hold the access token for a given user and connector, you're ready to perform any operation that falls under the granted permissions, as shown in **Figure 2**.

A common type of social network app "listens" to some feed and posts automatically on behalf of the same user. In this case, you grab the access token once and keep on running the app until the user revokes permissions. Once you hold the access token, you don't need to deal with authentication as long as the user behind the token is actually logged in to the social network on the computer.

Posting Status and Photos

Figure 7 shows the UI of the sample WPF app once the user is successfully logged in to Facebook and the access token is safely stored. The UI includes a text box and a button to post. As you can see, the UI also defines a button to browse and selects a JPEG image from the local disk.

The code in **Figure 8** shows how to post an update with an attached picture.

In **Figure 7** you see the update and picture posted to my timeline. Note that the name of the connector used is presented under the name of the author.

Next Up: JavaScript

In this article I built a WPF app to post to Facebook on behalf of a user.

To build a mobile app, you follow the same pattern discussed here. Note that the pattern is valid regardless of the mobile platform of choice. I've successfully built Windows Phone and Android apps in this same way. My next column will show how to use JavaScript to program Facebook.

DINO ESPOSITO is the author of "Architecting Mobile Solutions for the Enterprise" (Microsoft Press, 2012) and "Programming ASP.NET MVC 3" (Microsoft Press, 2011), and coauthor of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2008). Based in Italy, Esposito is a frequent speaker at industry events worldwide. Follow him on Twitter at twitter.com/despos.

THANKS to the following technical expert for reviewing this article: [Scott Densmore](#)

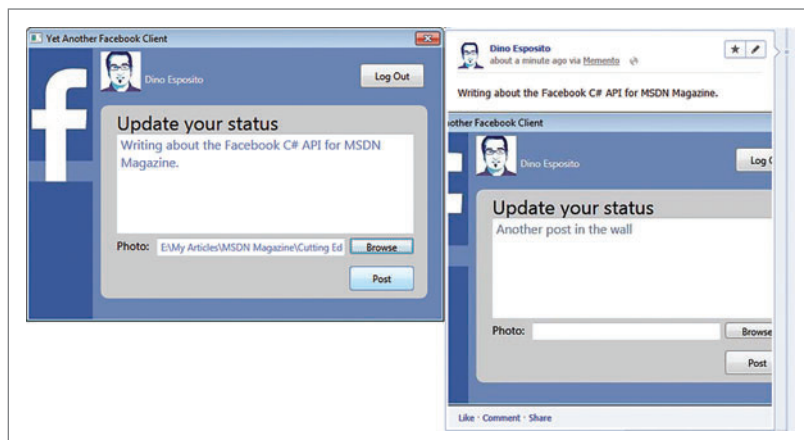


Figure 7 Posting from the Sample Client App

Globalize Your Business



Melissa Data can help you globalize your applications as you expand operations to other countries or reach new customers in emerging markets. As a world leading data quality vendor, we offer solutions to verify, correct and standardize addresses in over 240 countries. Eliminate returns, cut postage expenses, prevent fraud and keep your customers happy by verifying their address before you send a package.

- Reduce address correction fees – save up to \$10 per package
- Efficiently validate and correct addresses every time you ship
- Maintain high customer satisfaction

Accurate data. Delivered.

www.MelissaData.com/global
or call 1-800-MELISSA (635-4772)

- ✓ **Address Verification**
- ✓ **ID Verification**
- ✓ **Email Verification**
- ✓ **GeoCoding**
- ✓ **IP Location**
- ✓ **Name Parsing**
- ✓ **Phone Verification**
- ✓ **Record Matching**

MELISSA DATA®
Your Partner in Data Quality



The Evolution of Threads and I/O in Windows

When starting a new project, do you ask yourself whether your program will be compute-bound or I/O-bound? You should. I've found that in most cases it's either one or the other. You might be working on an analytics library that's handed a pile of data and keeps a bunch of processors busy while crunching it all down to a set of aggregates. Alternatively, your code might spend most of its time waiting for stuff to happen, for data to arrive over the network, a user to click on something or perform some complex six-fingered gesture. In this case, the threads in your program aren't doing very much. Sure, there are cases where programs are heavy both on I/O and computation. The SQL Server database engine comes to mind, but that's less typical of today's computer programming. More often than not, your program is tasked with coordinating the work of others. It might be a Web server or client communicating with a SQL database, pushing some computation to the GPU or presenting some content for the user to interact with. Given all of these different scenarios, how do you decide what threading capabilities your program requires and what concurrency building blocks are necessary or useful? Well, that's a tough question to answer generally and something you'll need to analyze as you approach a new project. It's helpful, however, to understand the evolution of threading in Windows and C++ so you can make an informed decision based on the practical choices that are available.

Your program is no more awesome if you use twice as many threads as another program. It's what you do with those threads that counts.

Threads, of course, provide no direct value whatsoever to the user. Your program is no more awesome if you use twice as many threads as another program. It's what you do with those threads that counts. To illustrate these ideas and the way in which threading has evolved over time, let me take the example of reading some data from a file. I'll skip over the C and C++ libraries because their support for I/O is mostly geared toward synchronous, or blocking, I/O, and this

is typically not of interest unless you're building a simple console program. Of course, there's nothing wrong with that. Some of my favorite programs are console programs that do one thing and do it really well. Still, that's not very interesting, so I'll move on.

One Thread

To begin with, I'll start with the Windows API and the good old—and even aptly named—`ReadFile` function. Before I can start reading the contents of a file, I need a handle to the file, and this handle is provided by the immensely powerful `CreateFile` function:

```
auto fn = L"C:\\data\\greeting.txt";
auto f = CreateFile(fn, GENERIC_READ, FILE_SHARE_READ, nullptr,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
ASSERT(f);
```

To keep the examples brief, I'll just use the `ASSERT` and `VERIFY` macros as placeholders to indicate where you'll need to add some error handling to manage any failures reported by the various API functions. In this code snippet, the `CreateFile` function is used to open rather than create the file. The same function is used for both operations. The *Create* in the name is more about the fact that a kernel file object is created, and not so much about whether or not a file is created in the file system. The parameters are pretty self-explanatory and not too relevant to this discussion, with the exception of the second-to-last, which lets you specify a set of flags and attributes indicating the type of I/O behavior you need from the kernel. In this case I used the `FILE_ATTRIBUTE_NORMAL` constant, which just indicates that the file should be opened for normal synchronous I/O. Remember to call the `CloseHandle` function to release the kernel's lock on the file when you're done with it. A handle wrapper class, such as the one I described in my July 2011 column, "C++ and the Windows API" (msdn.microsoft.com/magazine/hh288076), will do the trick.

I can now go ahead and call the `ReadFile` function to read the contents of the file into memory:

```
char b[64];
DWORD c;
VERIFY(ReadFile(f, b, sizeof(b), &c, nullptr));
printf("> %.*s\n", c, b);
```

As you might expect, the first parameter specifies the handle to the file. The next two describe the memory into which the file's contents should be read. `ReadFile` will also return the actual number of bytes copied should there be fewer bytes available than were requested. The final parameter is only used for asynchronous I/O, and I'll get back to that in a moment. In this simplistic example, I then simply print out the characters that were actually read from the file. Naturally, you might need to call `ReadFile` multiple times if required.

WPF lives!



➔ **XCEED Business Suite for WPF**

The essential set of WPF controls for all your line-of-business solutions. Includes the industry-leading **Xceed DataGrid for WPF**.
A total of 85 tools!

Two Threads

This model of I/O is simple to grasp and certainly quite useful for many small programs, particularly console-based programs. But it doesn't scale very well. If you need to read two separate files at the same time, perhaps to support multiple users, you'll need two threads. No problem—that's what the `CreateThread` function is for. Here's a simple example:

```
auto t = CreateThread(nullptr, 0, [] (void *) -> DWORD
{
    CreateFile/ReadFile/CloseHandle
    return 0;
},
nullptr, 0, nullptr);

ASSERT(t);
WaitForSingleObject(t, INFINITE);
```

Here I'm using a stateless lambda instead of a callback function to represent the thread procedure. The Visual C++ 2012 compiler conforms to the C++11 language specification in that stateless lambdas must be implicitly convertible to function pointers. This is convenient, and the Visual C++ compiler does one better by automatically producing the appropriate calling convention on the x86 architecture, which sports a variety of calling conventions.

The `CreateThread` function returns a handle representing a thread that I then wait on using the `WaitForSingleObject` function. The thread itself blocks while the file is read. In this way I can have multiple threads performing different I/O operations in tandem. I could then call `WaitForMultipleObjects` to wait until all of the threads have finished. Remember also to call `CloseHandle` to release the thread-related resources in the kernel.

This technique, however, doesn't scale beyond a handful of users or files or whatever the scalability vector is for your program. To be clear, it's not that multiple outstanding read operations don't scale. Quite the opposite. But it's the threading and synchronization overhead that will kill the program's scalability.

Back to One Thread

One solution to this problem is to use something called alertable I/O via asynchronous procedure calls (APCs). In this model your program relies on a queue of APCs that the kernel associates with each thread. APCs come in both kernel- and user-mode varieties. That is, the procedure, or function, that's queued might belong to a program in user mode, or even to some kernel-mode driver.

Figure 1 Alertable I/O with APCs

```
struct overlapped_buffer
{
    OVERLAPPED o;
    char b[64];
};

overlapped_buffer ob = {};

VERIFY(ReadFileEx(f, ob.b, sizeof(ob.b), &ob.o, [] (DWORD e, DWORD c,
OVERLAPPED * o)
{
    ASSERT(ERROR_SUCCESS == e);
    auto ob = reinterpret_cast<overlapped_buffer *>(o);

    printf("> %.*s\n", c, ob->b);
}));

SleepEx(INFINITE, true);
```

The latter is a simple way for the kernel to allow a driver to execute some code in the context of a thread's user-mode address space so that it has access to its virtual memory. But this trick is also available to user-mode programmers. Because I/O is fundamentally asynchronous in hardware anyway (and thus in the kernel), it makes sense to begin reading the file's contents and have the kernel queue an APC when it eventually completes.

If you need to read two separate files at the same time, perhaps to support multiple users, you'll need two threads.

To begin with, the flags and attributes passed to the `CreateFile` function must be updated to allow the file to provide overlapped I/O so that operations on the file aren't serialized by the kernel. The terms asynchronous and overlapped are used interchangeably in the Windows API, and they mean the same thing. Anyway, the `FILE_FLAG_OVERLAPPED` constant must be used when creating the file handle:

```
auto f = CreateFile(fn, GENERIC_READ, FILE_SHARE_READ, nullptr,
    OPEN_EXISTING, FILE_FLAG_OVERLAPPED, nullptr);
```

Again, the only difference in this code snippet is that I replaced the `FILE_ATTRIBUTE_NORMAL` constant with the `FILE_FLAG_OVERLAPPED` one, but the difference at run time is huge. To actually provide an APC that the kernel can queue at I/O completion, I need to use the alternative `ReadFileEx` function. Although `ReadFile` can be used to initiate asynchronous I/O, only `ReadFileEx` lets you provide an APC to call when it completes. The thread can then go ahead and do other useful work, perhaps starting additional asynchronous operations, while the I/O completes in the background.

Again, thanks to C++11 and Visual C++, a lambda can be used to represent the APC. The trick is that the APC will likely want to access the newly populated buffer, but this isn't one of the parameters to the APC, and because only stateless lambdas are allowed, you can't use the lambda to capture the buffer variable. The solution is to hang the buffer off the `OVERLAPPED` structure, so to speak. Because a pointer to the `OVERLAPPED` structure is available to the APC, you can then simply cast the result to a structure of your choice. **Figure 1** provides a simple example.

In addition to the `OVERLAPPED` pointer, the APC is also provided an error code as its first parameter and the number of bytes copied as its second. At some point, the I/O completes, but in order for the APC to run, the same thread must be placed in an alertable state. The simplest way to do that is with the `SleepEx` function, which wakes up the thread as soon as an APC is queued and executes any APCs before returning control. Of course, the thread may not be suspended at all if there are already APCs in the queue. You can also check the return value from `SleepEx` to find out what caused it to resume. You can even use a value of zero instead of `INFINITE` to flush the APC queue before proceeding without delay.



You used to think "Impossible"

Your Apps, Any Device

Now you think - game on!! The new tools in 12.2 help you envision and create engaging applications for the Web that can be accessed by mobile users on the go. And, with our Windows 8 XAML and JS tools you will begin to create highly interactive applications that address your customer needs today and build next generation touch enabled solutions for tomorrow.



Download your 30-day trial at
www.DevExpress.com

DXv2

The next generation of inspiring tools. **Today.**



Figure 2 The Completion Port Wrapper

```
class completion_port
{
    HANDLE h;

    completion_port(completion_port const &);
    completion_port & operator=(completion_port const &);

public:
    explicit completion_port(DWORD tc = 0) :
        h(CreateIoCompletionPort(INVALID_HANDLE_VALUE, nullptr, 0, tc))
    {
        ASSERT(h);
    }

    ~completion_port()
    {
        VERIFY(CloseHandle(h));
    }

    void add_file(HANDLE f, ULONG_PTR k = 0)
    {
        VERIFY(CreateIoCompletionPort(f, h, k, 0));
    }

    void queue(DWORD c, ULONG_PTR k, OVERLAPPED * o)
    {
        VERIFY(PostQueuedCompletionStatus(h, c, k, o));
    }

    void dequeue(DWORD & c, ULONG_PTR & k, OVERLAPPED *& o)
    {
        VERIFY(GetQueuedCompletionStatus(h, &c, &k, &o, INFINITE));
    }
};
```

Using SleepEx is not, however, all that useful and can easily lead to unscrupulous programmers to poll for APCs, and that's never a good idea. Chances are that if you're using asynchronous I/O from a single thread, this thread is also your program's message loop. Either way, you can also use the MsgWaitForMultipleObjectsEx function to wait for more than just APCs and build a more compelling single-threaded runtime for your program. The potential drawback with APCs is that they can introduce some challenging re-entrancy bugs, so do keep that in mind.

One Thread per Processor

As you find more things for your program to do, you might notice that the processor on which your program's thread is running is getting busier while the remaining processors on the computer are sitting around waiting for something to do. Although APCs are about the most efficient way to perform asynchronous I/O, they have the obvious drawback of only ever completing on the same thread that initiated the operation. The challenge then is to develop a solution that can scale this to all of the available processors. You might conceive of a design of your own doing, perhaps coordinating work among a number of threads with alertable message loops, but nothing you could implement would come close to the sheer performance and scalability of the I/O completion port, in large part because of its deep integration with different parts of the kernel.

While an APC allows asynchronous I/O operations to complete on a single thread, a completion port allows any thread to begin an I/O operation and have the results processed by an arbitrary thread. A completion port is a kernel object that you create before associating it with any number of file objects, sockets, pipes and

more. The completion port exposes a queuing interface whereby the kernel can push a completion packet onto the queue when I/O completes and your program can dequeue the packet on any available thread and process it as needed. You can even queue your own completion packets if needed. The main difficulty is getting around the confusing API. **Figure 2** provides a simple wrapper class for the completion port, making it clear how the functions are used and how they relate.

A completion port allows any thread to begin an I/O operation and have the results processed by an arbitrary thread.

The main confusion is around the double duty that the CreateIoCompletionPort function performs, first actually creating a completion port object and then associating it with an overlapped file object. The completion port is created once and then associated with any number of files. Technically, you can perform both steps in a single call, but this is only useful if you use the completion port with a single file, and where's the fun in that?

When creating the completion port, the only consideration is the last parameter indicating the thread count. This is the maximum number of threads that will be allowed to dequeue completion packets concurrently. Setting this to zero means that the kernel will allow one thread per processor.

Adding a file is technically called an association; the main thing to note is the parameter indicating the key to associate with the file. Because you can't hang extra information off the end of a handle like you can with an OVERLAPPED structure, the key provides a way for you to associate some program-specific information with

Figure 3 Thread Pool I/O

```
OVERLAPPED o = {};
char b[64];

auto io = CreateThreadpoolIo(f, [] (PTP_CALLBACK_INSTANCE, void * b,
    void *, ULONG e, ULONG_PTR c, PTP_IO)
{
    ASSERT(ERROR_SUCCESS == e);

    printf("> %s\n", c, static_cast<char *>(b));
},
    b, nullptr);

ASSERT(io);
StartThreadpoolIo(io);

auto r = ReadFile(f, b, sizeof(b), nullptr, &o);

if (!r && ERROR_IO_PENDING != GetLastError())
{
    CancelThreadpoolIo(io);
}

WaitForThreadpoolIoCallbacks(io, false);
CloseThreadpoolIo(io);
```


DEVELOPED FOR INTUITIVE USE

DynamicPDF—Comprehensive PDF Solutions for .NET Developers

ceTe Software's DynamicPDF products provide real-time PDF generation, manipulation, conversion, printing, viewing, and much more. Providing the best of both worlds, the object models are extremely flexible but still supply the rich features you need as a developer. Reliable and efficient, the high-performance software is easy to learn and use. If you do encounter a question with any of our components, simply contact ceTe Software's readily available, industry-leading support team.



DynamicPDF

[WWW.DYNAMICPDF.COM](http://www.DynamicPDF.com)



TRY OUR PDF SOLUTIONS FREE TODAY!

www.DynamicPDF.com/eval or call 800.631.5006 | +1 410.772.8620

ceTesoftware

the file. Whenever the kernel queues a completion packet related to this file, this key will also be included. This is particularly important because the file handle isn't even included in the completion packet.

As I said, you can queue your own completion packets. In this case, the values you provide are entirely up to you. The kernel doesn't care and won't try to interpret them in any way. Thus, you can provide a bogus OVERLAPPED pointer and the exact same address will be stored in the completion packet.

In most cases, however, you'll wait for the kernel to queue the completion packet once an asynchronous I/O operation completes. Typically a program creates one or more threads per processor and calls `GetQueuedCompletionStatus`, or my dequeue wrapper function, in an endless loop. You might queue a special control completion packet—one per thread—when your program needs to come to an end and you want these threads to terminate. As with APCs, you can hang more information off the OVERLAPPED structure to associate extra information with each I/O operation:

```
completion_port p;
p.add_file(f);
overlapped_buffer ob = {};
ReadFile(f, ob.b, sizeof(ob.b), nullptr, &ob.o);
```

Here I'm again using the original `ReadFile` function, but in this case I'm providing a pointer to the OVERLAPPED structure as its last parameter. A waiting thread might dequeue the completion packet as follows:

```
DWORD c;
ULONG_PTR k;
OVERLAPPED * o;
p.dequeue(c, k, o);
auto ob = reinterpret_cast<overlapped_buffer *>(o);
```

A Pool of Threads

If you've been following my column for some time, you'll recall that I spent five months last year covering the Windows thread pool in detail. It will also be no surprise to you that this same thread pool API is implemented using I/O completion ports, providing this same work-queuing model but without the need to manage the threads yourself. It also provides a host of features and conveniences that make it a compelling alternative to using the completion port object directly. If you haven't already done so, I encourage you to read those columns to get up to speed on the Windows thread pool API. A list of my online columns is available at bit.ly/StHJtH.

At a minimum, you can use the `TrySubmitThreadpoolCallback` function to get the thread pool to create one of its work objects internally and have the callback immediately submitted for execution. It doesn't get much simpler than this:

```
TrySubmitThreadpoolCallback([](PTP_CALLBACK_INSTANCE, void *)
{
    // Work goes here!
},
nullptr, nullptr);
```

If you need a bit more control, you can certainly create a work object directly and associate it with a thread pool environment and cleanup group. This will also give you the best possible performance.

Of course, this discussion is about overlapped I/O, and the thread pool provides I/O objects just for that. I won't spend much time on this, as I've already covered it in detail in my December 2011 column, "Thread Pool Timers and I/O" (msdn.microsoft.com/magazine/hh580731), but **Figure 3** provides a new example.

Given that `CreateThreadpoolIo` lets me pass an additional context parameter to the queued callback, I don't need to hang the buffer off the OVERLAPPED structure, although I could certainly do that if needed. The main things to keep in mind here are that `StartThreadpoolIo` must be called prior to beginning the asynchronous I/O operation, and `CancelThreadpoolIo` must be called should the I/O operation fail or complete inline, so to speak.

Taking the concept of a thread pool to new heights, the new Windows API for Windows Store apps also provides a thread pool abstraction.

Fast and Fluid Threads

Taking the concept of a thread pool to new heights, the new Windows API for Windows Store apps also provides a thread pool abstraction, although a much simpler one with far fewer features. Fortunately, nothing prevents you from using an alternative thread pool appropriate for your compiler and libraries. Whether you'll get it past the friendly Windows Store curators is another story. Still, the thread pool for Windows Store apps is worth mentioning, and it integrates the asynchronous pattern embodied by the Windows API for Windows Store apps.

Using the slick C++/CX extensions provides a relatively simple API for running some code asynchronously:

```
ThreadPool::RunAsync(ref new WorkItemHandler([] (IAsyncAction ^)
{
    // Work goes here!
}));
```

Syntactically this is pretty straightforward. We can even hope that this will become simpler in a future version of Visual C++ if the compiler can automatically generate a C++/CX delegate from a lambda—at least conceptually—the same as it does today for function pointers.

Still, this relatively simple syntax belies a great deal of complexity. At a high level, `ThreadPool` is a static class, to borrow a term from the C# language, and thus can't be created. It provides a few overloads of the static `RunAsync` method, and that's it. Each takes at least a delegate as its first parameter. Here I'm constructing the delegate with a lambda. The `RunAsync` methods also return an `IAsyncAction` interface, providing access to the asynchronous operation.

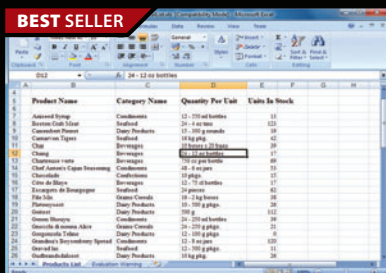
As a convenience, this works pretty well and integrates nicely into the asynchronous programming model that pervades the Windows API for Windows Store apps. You can, for example, wrap the `IAsyncAction` interface returned by the `RunAsync` method in a Parallel Patterns Library (PPL) task and achieve a level of composability similar to what I described in my September and October columns, "The Pursuit of Efficient and Composable Asynchronous Systems" (msdn.microsoft.com/magazine/jj618294) and "Back to the Future with Resumable Functions" (msdn.microsoft.com/magazine/jj658968).



Aspose.Total for .NET from \$2,449.02

Every Aspose .NET component in one package.

- Programmatically manage popular file formats including Word, Excel, PowerPoint and PDF
- Add charting, email, spell checking, barcode creation, OCR, diagramming, imaging, project management and file format management to your .NET applications
- Common uses also include mail merge, adding barcodes to documents, building dynamic Excel reports on the fly and extracting text from PDF files

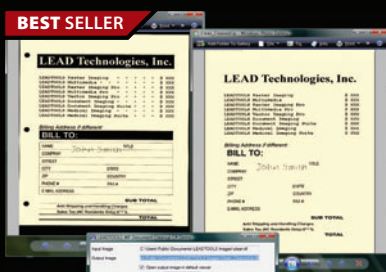


LEADTOOLS Document Imaging SDKs V17.5 from \$2,245.50



Add powerful document imaging functionality to desktop, tablet, mobile & web applications.

- Comprehensive document image cleanup, preprocessing, viewer controls and annotations
- Fast and accurate OCR, ICR and Forms Recognition with multi-threading support
- PDF & PDF/A Read / Write / Extract / View / Edit
- Barcode Detect / Read / Write for UPC, EAN, Code 128, Data Matrix, QR Code, PDF417
- Native WinRT Libraries for Windows Store & Zero-Footprint HTML5/JavaScript Controls



ComponentOne Studio Enterprise from \$1,315.60



.NET Tools for the Smart Developer: Windows, Web, and XAML.

- Hundreds of UI controls for all .NET platforms including grids, charts, reports and schedulers
- Supports Visual Studio 2012 and Windows 8
- Now includes Windows 8 Studios for WinRT XAML and WinJS
- New Cosmopolitan (Metro) theme provides a modern look and feel
- Royalty-free deployment and distribution



Help & Manual Professional from \$583.10



Easily create documentation for Windows, the Web and iPad.

- Powerful features in an easy accessible and intuitive user interface
- As easy to use as a word processor, but with all the power of a true WYSIWYG XML editor
- Single source, multi-channel publishing with conditional and customized output features
- Output to HTML, WebHelp, CHM, PDF, ePub, RTF, e-book or print
- Styles and Templates give you full design control



However, it's useful and somewhat sobering to realize what this seemingly innocuous code really represents. At the heart of the C++/CX extensions is a runtime based on COM and its IUnknown interface. Such an interface-based object model can't possibly provide static methods. There has to be an object for there to be an interface, and some sort of class factory to create that object, and indeed there is.

The Windows Runtime defines something called a runtime class that's very much akin to a traditional COM class. If you're old-school, you could even define the class in an IDL file and run it through a new version of the MIDL compiler specifically suited to the task, and it will generate .winmd metadata files and the appropriate headers.

A runtime class can have both instance methods and static methods. They're defined with separate interfaces. The interface containing the instance methods becomes the class's default interface, and the interface containing the static methods is attributed to the runtime class in the generated metadata. In this case the ThreadPool runtime class lacks the activatable attribute and has no default interface, but once created, the static interface can be queried for, and then those not-so-static methods may be called. **Figure 4** gives an example of what this might entail. Keep in mind that most of this would be compiler-generated, but it should give you a good idea of what it really costs to make that simple static method call to run a delegate asynchronously.

This is certainly a far cry from the relative simplicity and efficiency of calling the TrySubmitThreadPoolCallback function. It's helpful to understand the cost of the abstractions you use, even if you end up deciding that the cost is justified given some measure of productivity. Let me break it down briefly.

The WorkItemHandler delegate is actually an IUnknown-based IWorkItemHandler interface with a single Invoke method. The implementation of this interface isn't provided by the API but rather by the compiler. This makes sense because it provides a convenient

container for any variables captured by the lambda and the lambda's body would naturally reside within the compiler-generated Invoke method. In this example I'm simply relying on the Windows Runtime Library (WRL) RuntimeClass template class to implement IUnknown for me. I can then use the handy Make template function to create an instance of my WorkItemHandler. For stateless lambdas and functions pointers, I'd further expect the compiler to produce a static implementation with a no-op implementation of IUnknown to avoid the overhead of dynamic allocation.

As a C++ programmer, you can be thankful that Windows 8 isn't an entirely new platform and the traditional Windows API is still at your disposal, if and when you need it.

To create an instance of the runtime class, I need to call the RoGetActivationFactory function. However, it needs a class ID. Notice that this isn't the CLSID of traditional COM but rather the fully qualified name of the type, in this case, Windows.System.Threading.ThreadPool. Here I'm using a constant array generated by the MIDL compiler to avoid having to count the string at run time. As if that weren't enough, I need to also create an HSTRING version of this class ID. Here I'm using the WindowsCreateStringReference function, which, unlike the regular WindowsCreateString function, doesn't create a copy of the source string. As a convenience, WRL also provides the HStringReference class that wraps up this functionality. I can now call the RoGetActivationFactory function, requesting the IThreadPoolStatics interface directly and storing the resulting pointer in a WRL-provided smart pointer.

I can now finally call the RunAsync method on this interface, providing it with my IWorkItemHandler implementation as well as the address of an IAsyncAction smart pointer representing the resulting action object.

It's then perhaps not surprising that this thread pool API doesn't provide anywhere near the amount of functionality and flexibility provided by the core Windows thread pool API or the Concurrency Runtime. The benefit of C++/CX and the runtime classes is, however, realized along the boundaries between the program and the runtime itself. As a C++ programmer, you can be thankful that Windows 8 isn't an entirely new platform and the traditional Windows API is still at your disposal, if and when you need it. ■

KENNY KERR is a software craftsman with a passion for native Windows development. Reach him at kennykerr.ca.

THANKS to the following technical expert for reviewing this article:
James P. McNellis

Figure 4 The WinRT Thread Pool

```
class WorkItemHandler :
public RuntimeClass<RuntimeClassFlags<ClassicCom>,
IWorkItemHandler>
{
virtual HRESULT __stdcall Invoke(IAsyncAction *)
{
// Work goes here!

return S_OK;
}
};

auto handler = Make<WorkItemHandler>();

HSTRING_HEADER header;
HSTRING clsid;
auto hr = WindowsCreateStringReference(
RuntimeClass_Windows_System_Threading_ThreadPool,
_countof(RuntimeClass_Windows_System_Threading_ThreadPool)
- 1, &header, &clsid);
ASSERT(S_OK == hr);

ComPtr<IThreadPoolStatics> tp;
hr = RoGetActivationFactory(
clsid, __uuidof(IThreadPoolStatics),
reinterpret_cast<void **>(tp.GetAddressOf()));
ASSERT(S_OK == hr);

ComPtr<IAsyncAction> a;
hr = tp->RunAsync(handler.Get(), a.GetAddressOf());
ASSERT(S_OK == hr);
```

We didn't invent the Internet...

...but our components help you power the apps that bring it to business.



TOOLS • COMPONENTS • ENTERPRISE ADAPTERS

- **E-Business**
AS2, EDI/X12, NAESB, OFTP ...
- **Credit Card Processing**
Authorize.Net, TSYS, FDMS ...
- **Shipping & Tracking**
FedEx, UPS, USPS ...
- **Accounting & Banking**
QuickBooks, OFX ...
- **Internet Business**
Amazon, eBay, PayPal ...
- **Internet Protocols**
FTP, SMTP, IMAP, POP, WebDav ...
- **Secure Connectivity**
SSH, SFTP, SSL, Certificates ...
- **Secure Email**
S/MIME, OpenPGP ...
- **Network Management**
SNMP, MIB, LDAP, Monitoring ...
- **Compression & Encryption**
Zip, Gzip, Jar, AES ...



The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to

any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

connectivity
powered by 

To learn more please visit our website →

www.nsoftware.com



Shrink EF Models with DDD Bounded Contexts

When defining models for use with the Entity Framework (EF), developers often include all of the classes to be used throughout the entire application. This might be a result of creating a new Database First model in the EF Designer and selecting all available tables and views from the database. For those of you using Code First to define your model, it might mean creating DbSet properties in a single DbContext for all of your classes or even unknowingly including classes that are related to those you've targeted.

When you're working with a large model and a large application, there are numerous benefits to designing smaller, more-compact models that are targeted to specific application tasks, rather than having a single model for the entire solution. In this column, I'll introduce you to a concept from domain-driven design (DDD)—Bounded Context—and show you how to apply it to build a targeted model with EF, focusing on doing this with the greater flexibility of the EF Code First feature. If you're new to DDD, this a great approach to learn even if you aren't committing fully to DDD. And if you're already using DDD, you'll benefit by seeing how you can use EF while following DDD practices.

Domain-Driven Design and Bounded Context

DDD is a fairly large topic that embraces a holistic view of software design. Paul Rayner, who teaches DDD workshops for Domain Language (DomainLanguage.com), puts it succinctly:

"DDD advocates pragmatic, holistic and continuous software design: collaborating with domain experts to embed rich domain models in the software—models that help solve important, complex business problems."

DDD includes numerous software design patterns, one of which—Bounded Context—lends itself perfectly to working with EF. Bounded Context focuses on developing small models that target supporting specific operations in your business domain. In his book, "Domain-Driven Design" (Addison Wesley, 2003), Eric Evans explains that Bounded Context "delimits the applicability of a particular model. Bounding contexts gives team members a clear and shared understanding of what has to be consistent and what can develop independently."

This article discusses the Entity Framework Power Tools beta 2. All information is subject to change.

Code download available at archive.msdn.microsoft.com/mag201301DataPoints.

Smaller models provide many benefits, allowing teams to define clear boundaries relating to design and development responsibilities. They also lead to better maintainability—because a context has a smaller surface area, you have fewer side effects to worry about when making modifications. Furthermore, there's a performance benefit when EF creates in-memory metadata for a model when it's first loaded into memory.

DDD is a fairly large topic that embraces a holistic view of software design.

Because I'm building bounded contexts with EF DbContext, I've referred to my DbContexts as "bounded DbContexts." However, the two are not actually equivalent: DbContext is a class implementation whereas Bounded Context encompasses the larger concept within the complete design process. I'll therefore refer to my DbContexts as "constrained" or "focused."

Comparing a Typical EF DbContext to Bounded Context

While DDD is most commonly applied to large application development in complex business domains, smaller apps can also benefit

Figure 1 Typical DbContext Containing All Domain Classes in the Solution

```
public class CompanyContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<SalaryHistory> SalaryHistories { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<LineItem> LineItems { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Shipment> Shipments { get; set; }
    public DbSet<Shipper> Shippers { get; set; }
    public DbSet<ShippingAddress> ShippingAddresses { get; set; }
    public DbSet<Payment> Payments { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Promotion> Promotions { get; set; }
    public DbSet<Return> Returns { get; set; }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Config specifies a 1:0..1 relationship between Customer and ShippingAddress
        modelBuilder.Configurations.Add(new ShippingAddressMap());
    }
}
```


Does your Team do more than just track bugs?

Free Trial and Single User FreePack™ available at www.alexcorp.com

Alexsys Team® does! Alexsys Team 2 is a multi-user Team management system that provides a powerful yet easy way to manage all the members of your team and their tasks - including defect tracking. Use Team right out of the box or tailor it to your needs.



Alexsys Team

Track all your project tasks in one database so you can work together to get projects done.

- Quality Control / Compliance Tracking
- Project Management
- End User Accessible Service Desk Portal
- Bugs and Features
- Action Items
- Sales and Marketing
- Help Desk

Native Smart Card Login Support including Government and DOD



New in Team 2.11

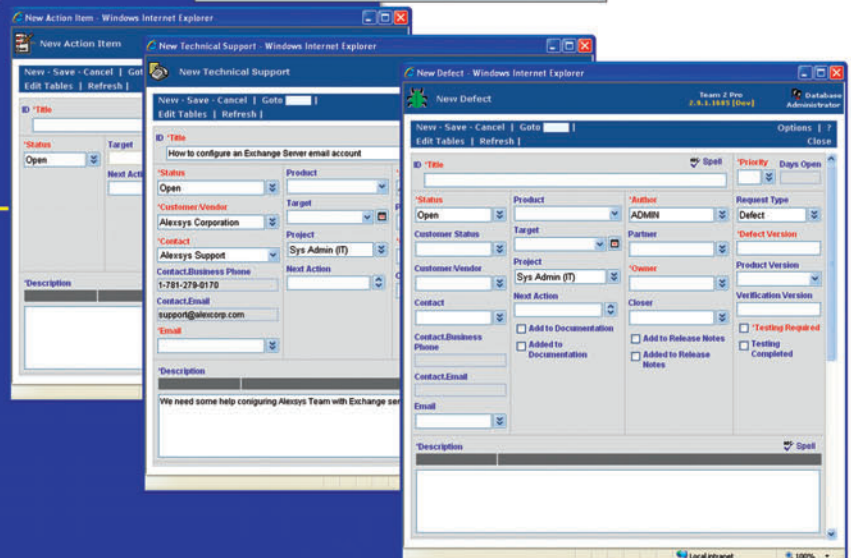
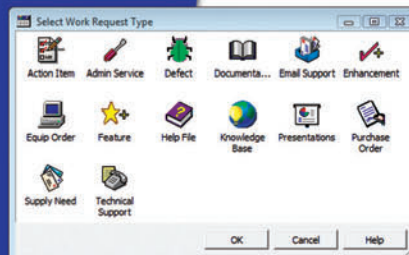
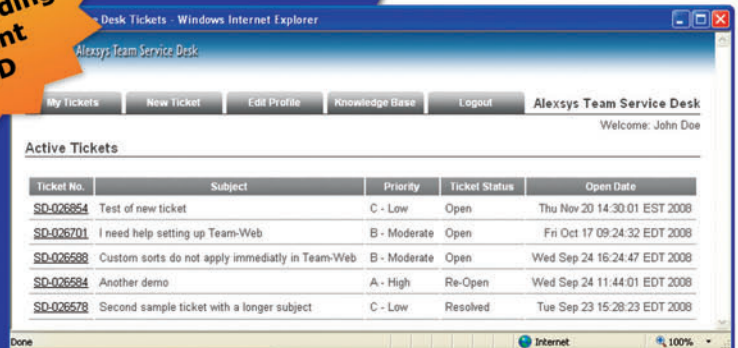
- Full Windows 7 Support
- Windows Single Sign-on
- System Audit Log
- Trend Analysis
- Alternate Display Fields for Data Normalization
- Lookup Table Filters
- XML Export
- Network Optimized for Enterprise Deployment

Service Desk Features

- Fully Secure
- Unlimited Users Self Registered or Active Directory
- Integrated into Your Web Site
- Fast/AJAX Dynamic Content
- Unlimited Service Desks
- Visual Service Desk Builder

Team 2 Features

- Windows and Web Clients
- Multiple Work Request Forms
- Customizable Database
- Point and Click Workflows
- Role Based Security
- Clear Text Database
- Project Trees
- Time Recording
- Notifications and Escalations
- Outlook Integration



Free Trial and Single User FreePack™ available at www.alexcorp.com. FreePack™ includes a free single user Team Pro and Team-Web license. Need more help? Give us a call at 1-888-880-ALEX (2539).

Team 2 works with its own standard database, while Team Pro works with Microsoft SQL, MySQL, and Oracle Servers.
Team 2 works with Windows 7/2008/2003/Vista/XP.
Team-Web works with Internet Explorer, Firefox, Netscape, Safari, and Chrome.

from many of its lessons. For the sake of this explanation, I'll focus on an application targeted to a specific subdomain: tracking sales and marketing for a company. Objects involved in this application might range from customers, orders and line items, to products, marketing, salespeople and even employees. Typically a DbContext would be defined to contain DbSet properties for every class in the solution that needs to be persisted to the database, as shown in **Figure 1**.

Imagine if this were a much more far-reaching application with hundreds of classes. And you might also have Fluent API configurations for some of those classes. That makes for an awful lot of code to wade through and manage in a single class. With such a large application, development might be divided up among teams. With this single company-wide DbContext, every team would need a subset of the code base that extends beyond their responsibilities. And any team's changes to this context could affect another team's work.

There are interesting questions you could ask yourself about this unfocused, overarching DbContext. For example, in the area of the application that targets the marketing department, do users have any need to work with employee salary history data? Does the shipping department need to access the same level of detail about a customer as a customer service agent? Would someone in the shipping department need to edit a customer record? For most common scenarios, the answer to these questions would generally be no, and this might help you see why it could make sense to have several DbContexts that manage smaller sets of domain objects.

A Focused DbContext for the Shipping Department

As DDD recommends working with smaller, more-focused models with well-defined context boundaries, let's narrow the scope of this DbContext to target shipping department functions and only those classes needed to perform the relevant tasks. Therefore, you can remove some DbSet properties from the DbContext, leaving only those you'll need to support business capabilities related to shipping. I've taken out Returns, Promotions, Categories, Payments, Employees and SalaryHistories:

```
public class ShippingDeptContext : DbContext
{
    public DbSet<Shipment> Shipments { get; set; }
    public DbSet<Shipper> Shippers { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<ShippingAddress> ShippingAddresses { get; set; }
    public DbSet<Order> Order { get; set; }
    public DbSet<LineItem> LineItems { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

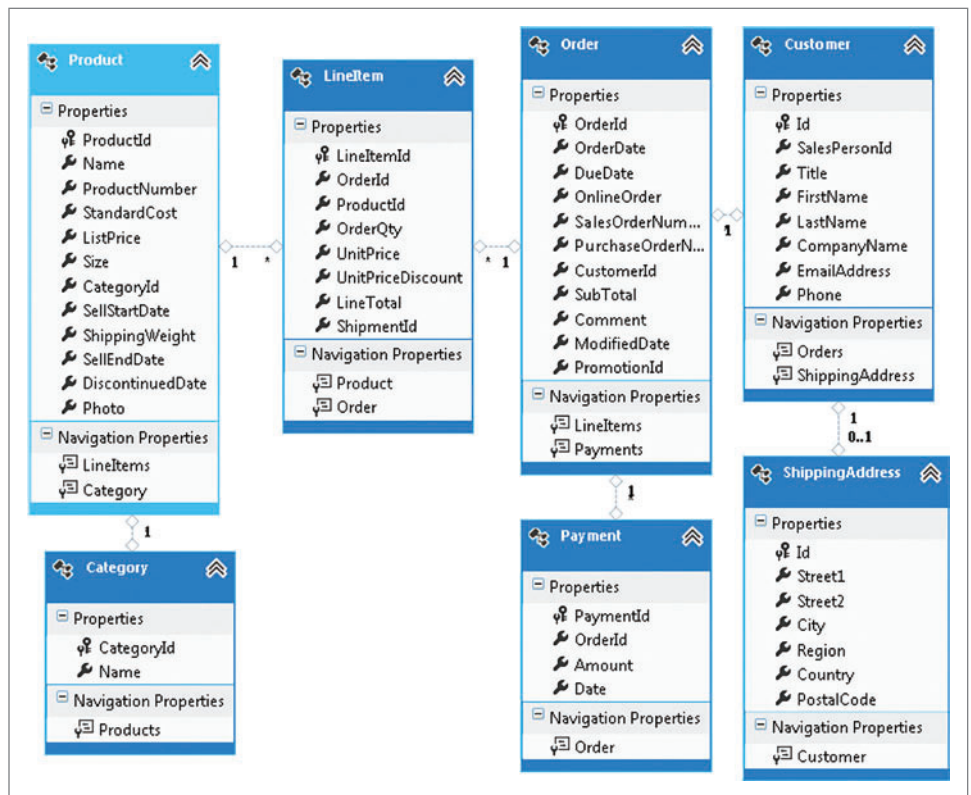


Figure 2 Visualized Model from First Pass at the ShippingContext

EF Code First uses the details of ShippingContext to infer the model. **Figure 2** shows a visualization of the model that will be created from this class, which I generated using the Entity Framework Power Tools beta 2. Now, let's start fine-tuning the model.

Tuning the DbContext and Creating More-Targeted Classes

There are still more classes involved in the model than I specified for shipping. By convention, Code First includes all classes that are reachable by other classes in the model. That's why Category and Payment showed up even though I removed their DbSet properties. So I'll tell the DbContext to ignore Category and Payment:

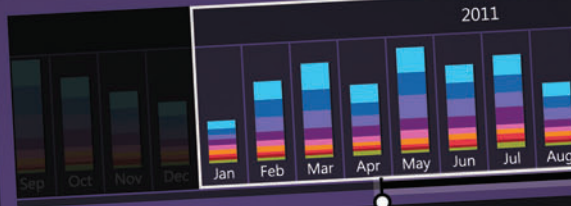
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Ignore<Category>();
    modelBuilder.Ignore<Payment>();
    modelBuilder.Configurations.Add(new ShippingAddressMap());
}
```

This ensures that Category and Payment don't get pulled into the model just because they're related to Product and Order.

You can refine this DbContext class even more without affecting the resulting model. With these DbSet properties, it's possible to explicitly query for each of these seven sets of data in your app. But if you think about the classes and their relationships, you might conclude that in this context it will never be necessary to query for the ShippingAddress directly—it can always be retrieved along with the Customer data. Even with no ShippingAddresses DbSet, you can rely on the same convention that automatically pulled in Category and Payment to pull ShippingAddress into the model because of its relationship to Customer. So you can remove the



Expense Dashboard January 1 - December 31, 2011



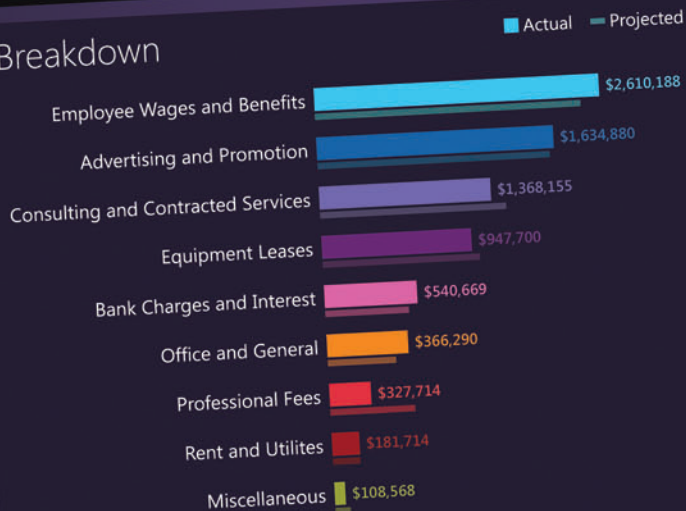
Total Expenses

\$9,221,481

From Target

▲ \$6,143,435
(5%)

Breakdown



Sales Analysis



ShippingAddresses property without losing the database mappings to ShippingAddress. You might be able to justify removing others, but let's focus on just this one:

```
public class ShippingContext : DbContext
{
    public DbSet<Shipment> Shipments { get; set; }
    public DbSet<Shipper> Shippers { get; set; }
    public DbSet<Customer> Customers { get; set; }
    // Public DbSet<ShippingAddress> ShippingAddresses { get; set; }
    public DbSet<Order> Order { get; set; }
    public DbSet<LineItem> LineItems { get; set; }
    public DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    { ... }
}
```

Within the context of processing shipments, I don't really need a full Customer object, a full Order object or a full LineItem object. I need only the Product to be shipped, the quantity (from LineItem), the Customer's name and ShippingAddress and any notes that might be attached to the Customer or to the Order. I'll have my DBA create a view that will return unshipped items—those with ShipmentId=0 or null. In the meantime, I can define a streamlined class that will map to that view with the relevant properties I anticipate needing:

```
[Table("ItemsToBeShipped")]
public class ItemToBeShipped
{
    [Key]
    public int LineItemId { get; set; }
    public int OrderId { get; set; }
    public int ProductId { get; set; }
    public int OrderQty { get; private set; }
    public OrderShippingDetail OrderShippingDetails { get; set; }
}
```

The logic of processing shipments requires querying for the ItemToBeShipped and then getting whatever Order details I might need along with the Customer and ShippingAddress. I could reduce my DbContext definition to allow me to query a graph starting with this new type and including the Order, Customer and ShippingAddress. However, because I know EF would achieve this with a SQL query designed to flatten the results and return repeated Order, Customer and ShippingAddress data along with each line item, I'll let the programmer query for an order and bring back a graph with the Customer and ShippingAddress. But again, I don't need all of the columns from the Order table, so I'll create a class that's better targeted for the shipping department, including information that could be printed on a shipping manifest. The class is the OrderShippingDetail class shown in **Figure 3**.

Notice that my ItemToBeShipped class has a navigation property for OrderShippingDetail and OrderShippingDetail has one for Customer. The navigation properties will help me with graphs when querying and saving.

There's one more piece to this puzzle. The shipping department will need to denote items as shipped and the LineItems table has a ShipmentId column that's used to tie a line item to a shipment. The app will need to update that ShipmentId field when an item is shipped. I'll create a simple class to take care of this task, rather than relying on the LineItem class that's used for sales:

```
[Table("LineItems")]
public class LineItemShipment
{
    [Key]
    public int LineItemId { get; set; }
    public int ShipmentId { get; set; }
}
```

Figure 3 The OrderShippingDetail Class

```
[Table("Orders")]
public class OrderShippingDetail
{
    [Key]
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public Nullable<DateTime> DueDate { get; set; }
    public string SalesOrderNumber { get; set; }
    public string PurchaseOrderNumber { get; set; }
    public Customer Customer { get; set; }
    public int CustomerId { get; set; }
    public string Comment { get; set; }
    public ICollection<ItemToBeShipped> OpenLineItems { get; set; }
}
```

Whenever an item has been shipped, you can create a new instance of this class with the proper values and force the application to update the LineItem in the database. It will be important to design your application to use the class only for this purpose. If you attempt to insert a LineItem using this class instead of one that accounts for non-nullable table fields such as OrderId, the database will throw an exception.

After some more fine-tuning, my ShippingContext is now defined as:

```
public class ShippingContext : DbContext
{
    public DbSet<Shipment> Shipments { get; set; }
    public DbSet<Shipper> Shippers { get; set; }
    public DbSet<OrderShippingDetail> Order { get; set; }
    public DbSet<ItemToBeShipped> ItemsToBeShipped { get; set; }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<LineItem>();
        modelBuilder.Ignore<Order>();
        modelBuilder.Configurations.Add(new ShippingAddressMap());
    }
}
```

Using the Entity Framework Power Tools beta 2 again to create an EDMX, I can see in the Model Browser window (**Figure 4**) that Code First infers that the model contains the four classes specified by the DbSet, as well as Customer and ShippingAddress, which were discovered by way of navigation properties from the OrderShippingDetail class.

Focused DbContext and Database Initialization

When using a smaller DbContext that supports specific Bounded Contexts in your application, it's critical to keep in mind two EF Code First default behaviors with respect to database initialization.

The first is that Code First will look for a database with the name of the context. That's not desirable when your application has a ShippingContext, a CustomerContext, a SalesContext and others. Instead, you want all DbContexts to point to the same database.

The second default behavior to consider is that Code First will use the model inferred by a DbContext to define the database schema. But now you have a DbContext that represents only a slice of the database. For

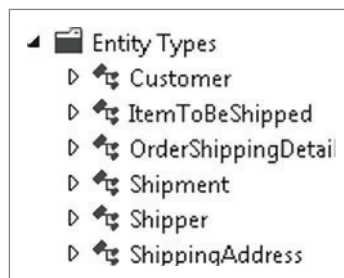


Figure 4 Model Browser View of ShippingContext Entities as Inferred by Code First

this reason, you don't want the DbContext classes to trigger database initialization.

It's possible to solve both of these problems in the constructor of each context class. For example, here in the ShippingContext class you can have the constructor specify the DPSalesDatabase and disable database initialization:

```
public ShippingContext() : base("DPSalesDatabase")
{
    Database.SetInitializer<ShippingContext>(null);
}
```

However, if you have a lot of DbContext classes in your app, this will become a maintenance problem. A better pattern is to specify a base class that disables database initialization and sets the database at the same time:

```
public class BaseContext<TContext>
    DbContext where TContext : DbContext
{
    static BaseContext()
    {
        Database.SetInitializer<TContext>(null);
    }
    protected BaseContext() : base("DPSalesDatabase")
    {}
}
```

with Customer and Shipping data for orders with unshipped line items.

- Retrieve an unshipped line item and create a new shipment. Set the shipment to the line item and insert the new shipment into the database while updating the line item in the database with the new shipment's key value.

These are the most critical functions you'd need to perform for shipping products to customers who have ordered them. The tests all pass, verifying that my DbContext works as expected. The tests are included in the sample download for this article.

Wrapping Up

I've not only created a DbContext that's focused specifically on supporting the shipping tasks, but thinking it through also helped me create more-efficient domain objects to be used with these tasks. Using DbContext to align my Bounded Context with my shipping subdomain in this way

DDD includes numerous software design patterns

Now my various context classes can implement the BaseContext instead of each having its own constructor:

```
public class
    ShippingContext:BaseContext<ShippingContext>
```

If you're doing new development and you want to let Code First create or migrate your database based on your classes, you'll need to create an "uber-model" using a DbContext that includes all of the classes and relationships needed to build a complete model that represents the database. However, this context must not inherit from BaseContext. When you make changes to your class structures, you can run some code that uses the uber-context to perform database initialization whether you're creating or migrating the database.

Exercising the Focused DbContext

With all of this in place, I created some automated integration tests to perform the following tasks:

- Retrieve open line items.
- Retrieve OrderShippingDetails along

means I don't have to wade through a quagmire of code to work on the shipping department feature of the application, and I can do what I need to the specialized domain objects without affecting other areas of the development effort.

You can see other examples of focusing DbContext using the DDD concept of Bounded Context in the book, "Programming Entity Framework: DbContext" (O'Reilly Media, 2011), which I coauthored with Rowan Miller. ■

JULIE LERMAN is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other Microsoft .NET topics at user groups and conferences around the world. She blogs at thedatafarm.com/blog and is the author of "Programming Entity Framework" (2010) as well as a Code First edition (2011) and a DbContext edition (2012), all from O'Reilly Media. Follow her on Twitter at twitter.com/julielerman.

THANKS to the following technical expert for reviewing this article: Paul Rayner

MSDN Magazine Online



It's like **MSDN Magazine**—only better. In addition to all the great articles from the print edition, you get:

- Code Downloads
- The MSDN Magazine Blog
- Digital Magazine Downloads
- Searchable Content

All of this and more at msdn.microsoft.com/magazine

msdn
magazine



Windows Azure Web Sites: Quick-and-Easy Hosting as a Service

Windows Azure Web Sites (WAWS) is a compelling feature of the Microsoft cloud platform. WAWS is a service built on top of the Windows Azure Platform as a Service (PaaS) framework that simplifies the deployment, management and scalability tasks associated with hosting cloud applications. If this makes you think WAWS is just another glorified hosting solution, think again. What's different is the level of scalability and high availability it provides, backed by multiple datacenters around the world and the failover mechanisms inherited from the PaaS components. Moreover, the Windows Azure platform offers many additional features and capabilities beyond WAWS, allowing developers to enhance their applications as their needs grow, including support for big data, identity, mobile services and more.

Datacenter Deployment Models

The taxonomy shown in **Figure 1** illustrates the different models associated with the deployment of a Web application in a public datacenter. The number of layers delegated to the platform or vendor lets us distinguish the terms Infrastructure as a Service (IaaS), PaaS and Software as a Service (SaaS).

The big difference compared with the traditional PaaS model offered by Windows Azure is that some of the deployment and configuration tasks are greatly simplified.

The more layers you delegate to the vendor, the easier it becomes to deploy a solution online, though generally the level of customization for the delegated layers decreases. Windows Azure offers both IaaS and PaaS deployment models that, combined with other services, allow companies to create sophisticated and highly scalable cloud architectures. Based on this, where do we place WAWS?

Code download available at msdnmagazine.azurewebsites.net/Person.

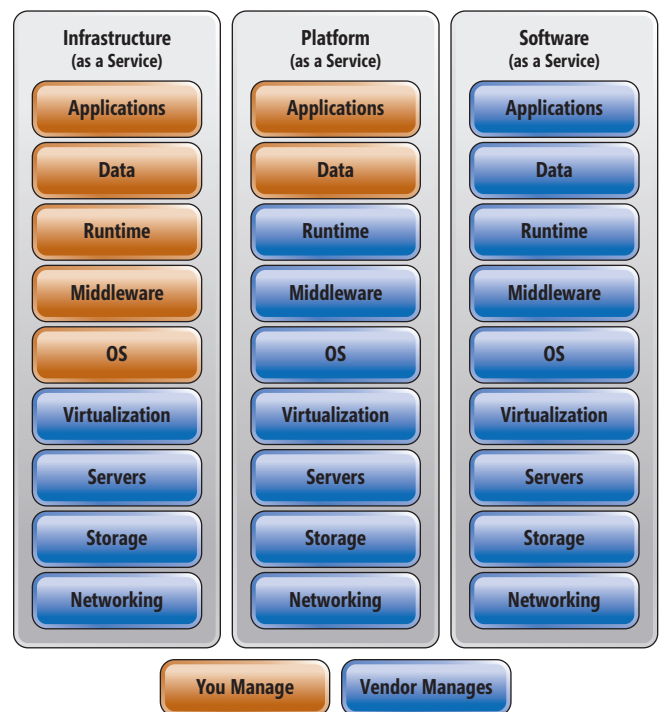


Figure 1 Taxonomy for Cloud Deployment Models

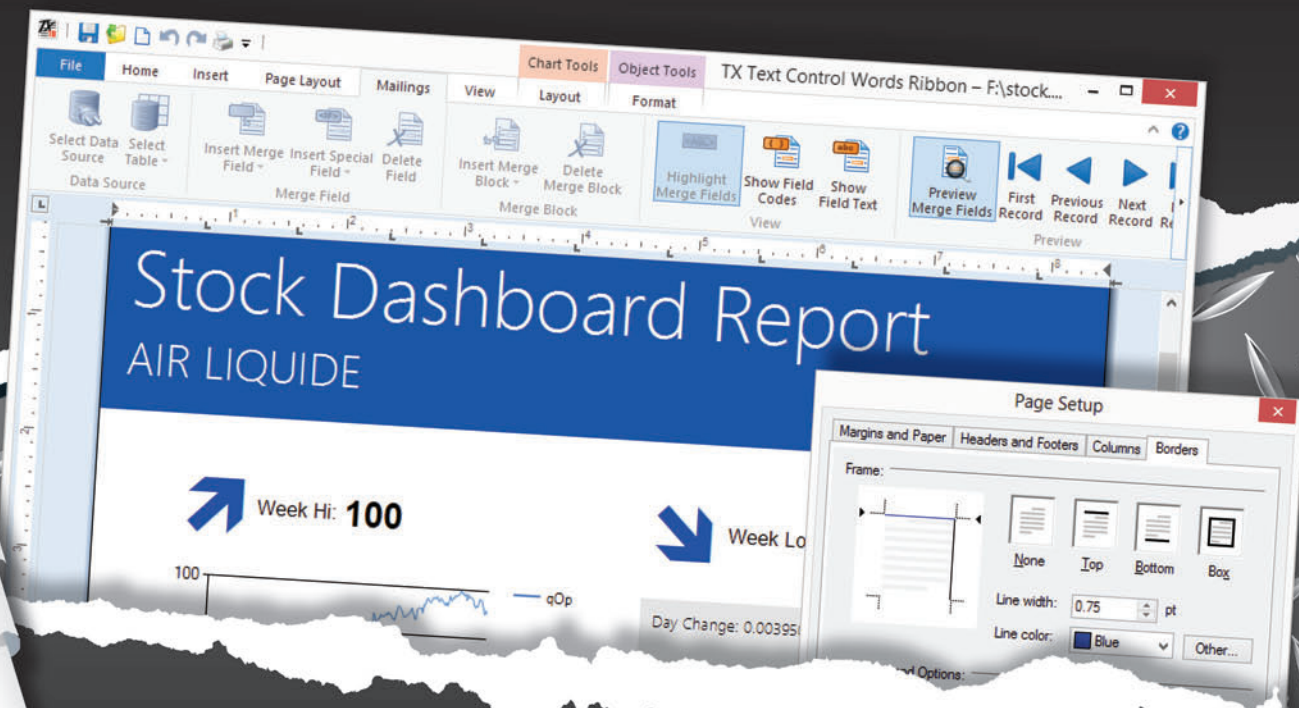
Why You'll Love Windows Azure Web Sites

As we mentioned, WAWS is built on top of the Windows Azure PaaS framework, which means you only have to worry about maintaining the application and data layers of the deployment stack. The big difference compared with the traditional PaaS model offered by Windows Azure is that some of the deployment and configuration tasks are greatly simplified. This is possible thanks to the provisioning process underlying WAWS, which is based on a combination of stateless worker roles, storage Binary Large Objects (BLOBs) and relational databases.

Figure 2 shows the architecture that supports WAWS. This is an elegant and robust approach to providing a multi-tenant environment for hosting Web applications. The way it works is simple:

1. A client makes a request to host <http://foo.azurewebsites.net>.
2. The request goes through the Windows Azure Network Load Balancer, reaching the appropriate deployment.
3. The Application Request Routing (ARR) module gets info from the runtime database about foo.com,

FLOW TYPE LAYOUT REPORTING



Reuse MS Word documents or templates as your reporting templates.



Easy database connection with master-detail nested blocks.



Powerful, programmable template designer with full sources for Visual Studio®.



Integrate dynamic 2D and 3D charting to your reports.



Create print-ready, digitally signed Adobe PDF and PDF/A documents.



Create flow type layouts with tables, columns, images, headers and footers and more.

TX
TEXTCONTROL®
word processing components



Visual Studio

Microsoft

Partner

US +1 877-462-4772

EU +49 421-4270671-0

WWW.TEXTCONTROL.COM

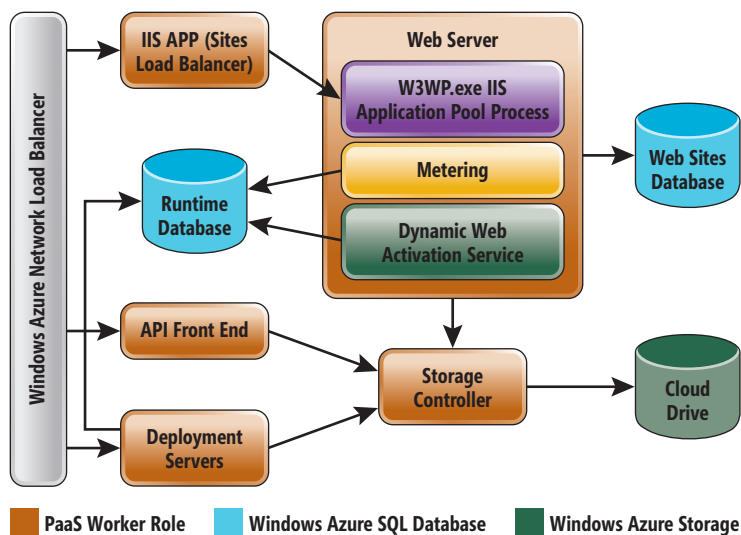


Figure 2 Windows Azure Web Sites Under the Hood

determines which Web servers should host or are currently hosting the Web site, and forwards the request to the corresponding one.

4. The request is processed by the Web server, aided by the storage controller to reach the specific instance of SQL Server in case data is being read or written.
5. Once the request has been processed, a response is sent back to the client.

Inside each of the Web servers, multiple W3WP.exe processes— or sites—are hosted, monitored by a specific process that oversees resource consumption, errors and logging. Also, the Dynamic Web Activation Service is responsible for site activation and deactivation, quota enforcement and deployment of the application files. Two additional components complete the picture: the deployment servers host the FTP and Web Deploy engines, while the API front end provides an interface for automation purposes.

This architecture is what provides a near-instant deployment experience, with the option of scaling out when required.

There are many great reasons to consider WAWS. One is that this service provides strong support for client-side markup and scripting, including built-in connectivity to SQL Server and MySQL databases. This means you don't need to worry about implementing back-end Web services and server logic. Another good reason to consider WAWS is its ability to scale up and down to support Web users in both the busiest and quietest of times.

Because you pay for only what you use, you can provide a responsive UX while keeping costs low. The WAWS service also supports continuous development methodologies, so you can deploy directly from

source code repositories, using Git on a client, GitHub, CodePlex, Bitbucket or Team Foundation Server (TFS). It's possible to instantly upload new code or continuously integrate with an online repository. Thanks to its provisioning model, the WAWS service offers built-in support for WordPress, Joomla!, Drupal, DotNetNuke, Umbraco and more. These templates are categorized in the WAWS gallery, and include blogs, content management, ecommerce, forums and wikis.

Keeping It Real

As with any technology that simplifies a problem, there are always trade-offs. For instance, WAWS doesn't support a staging environment for testing, or network isolation to bridge cloud applications with on-premises networks (including Windows Azure Connect or Windows Azure Virtual Networks). However, users can choose to use the Service Bus Relay service to bridge the gap with corporate resources. In addition, you can't run a remote desktop to the compute instances, or kick off a startup task during your

Because you pay for only what you use, you can provide a responsive UX while keeping costs low.

deployment. Even so, WAWS is a perfect solution for the development and deployment of Web applications that don't require a high level of customization.

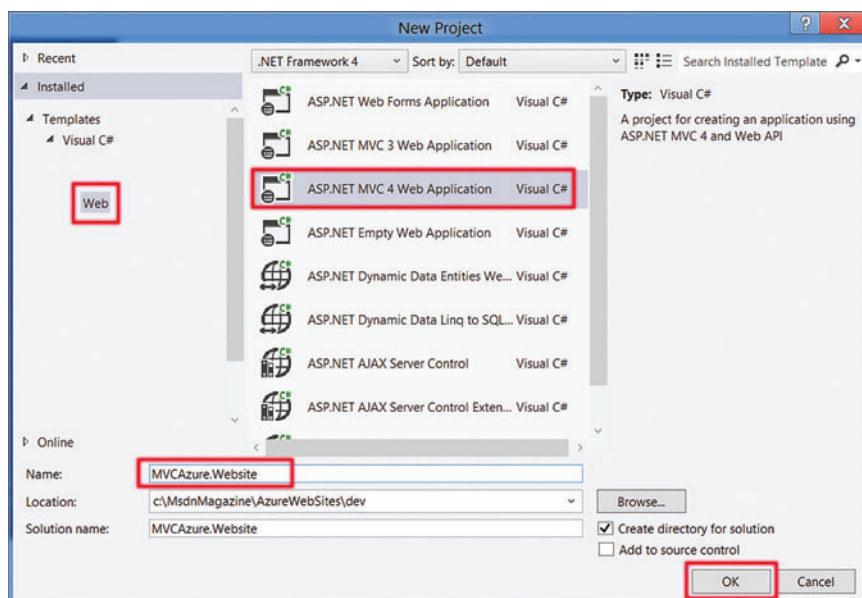


Figure 3 Creating a New ASP.NET MVC 4 Web Application Project

SpreadsheetGear

Performance Spreadsheet Components

SpreadsheetGear 2012 Now Available

NEW!

WPF and Silverlight controls, multithreaded recalc, 64 new Excel compatible functions, save to XPS, improved efficiency and performance, Windows 8 support, Windows Server 2012 support, Visual Studio 2012 support and more.

Excel Reporting for ASP.NET, WinForms, WPF and Silverlight



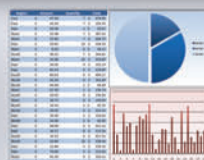
Easily create richly formatted Excel reports without Excel from any ASP.NET, Windows Forms, WPF or Silverlight application using spreadsheet technology built from the ground up for performance, scalability and reliability.

Excel Compatible Windows Forms, WPF and Silverlight Controls



Add powerful Excel compatible viewing, editing, formatting, calculating, filtering, charting, printing and more to your Windows Forms, WPF and Silverlight applications with the easy to use WorkbookView controls.

Excel Dashboards, Calculations, Charting and More



You and your users can design dashboards, reports, charts, and models in Excel or the SpreadsheetGear Workbook Designer rather than hard to learn developer tools and you can easily deploy them with one line of code.

**Free
30 Day
Trial**

Download our fully functional 30-Day evaluation and bring Excel Reporting, Excel compatible charting, Excel compatible calculations and much more to your ASP.NET, Windows Forms, WPF, Silverlight and other Microsoft .NET Framework solutions.

www.SpreadsheetGear.com



SpreadsheetGear

Toll Free USA (888) 774-3273 | Phone (913) 390-4797 | sales@spreadsheetgear.com

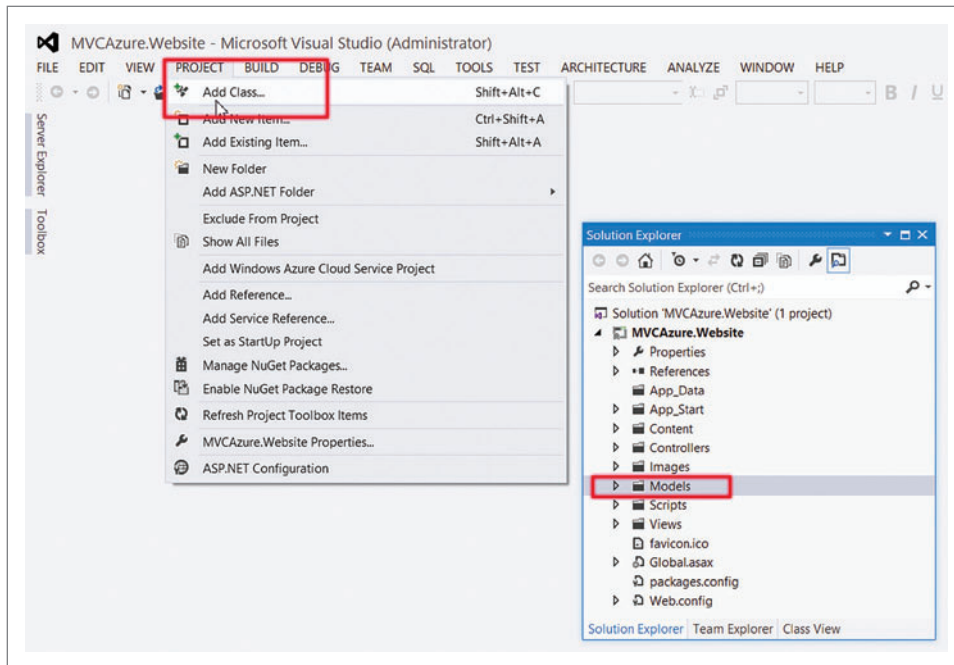


Figure 4 Adding a Class to the Models Folder

Hands On

Now let's walk through a simple tutorial. As we do, you'll begin to realize you don't need to learn anything new about the cloud. You can focus on creating your Web app and its data just as you always have. The details of moving to the cloud are abstracted away by the service, providing a frictionless deployment experience. You can even write the same classic ASP code you built in 1996 and run it in WAWS.

We'll explore the basic elements of the WAWS service by creating a simple yet practical ASP.NET MVC 4 application using SQL Server as the back end. This simple application will support Create, Read, Update and Delete (CRUD) operations and will require hardly any code. When we're through provisioning the WAWS and building the ASP.NET MVC 4 application, we'll deploy it using Web Deploy from Visual Studio 2012. The only expense is the SQL Database—

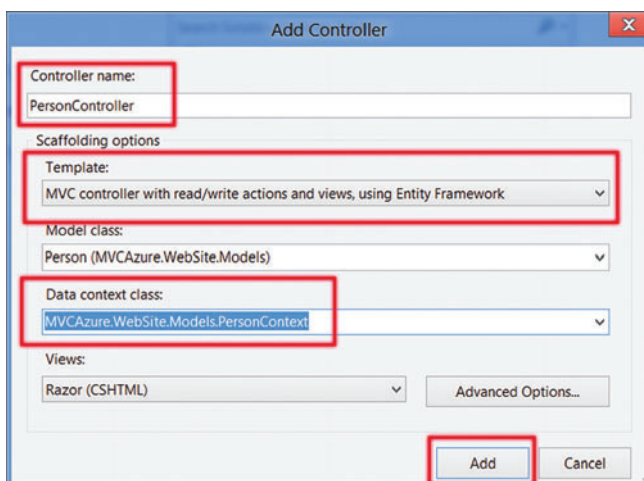


Figure 5 Adding an MVC Controller Class

running the ASP.NET MVC 4 application within WAWS is free. You can run up to 10 sites for free with no expiration date.

Start Visual Studio and Create an MVC Project

To start, open Visual Studio 2012 as Administrator and click File | New | Project. Choose Web from the Installed Templates, then select ASP.NET MVC 4 Web Application. Name the project MVCAzure.Website and click OK (see Figure 3). You'll need to select a Project Template next. Choose Internet Application here. The View Engine will be Razor.

Code First

A Code First approach means that the classes you create in Visual Studio will define the underlying

tables in SQL Server. In an ASP.NET MVC 4 app, this means we'll add some class modules under the Models folder in Visual Studio Solution Explorer, as shown in Figure 4. The properties defined in the class modules will become SQL Database table structures. This approach is productive because you don't need to worry about mapping your C#/Visual Basic objects to relational data structures. We'll leverage the Entity Framework to simplify this object-relational mapping. The Entity Framework is an object-relational mapper that makes interacting with a relational database much easier. It allows you to perform CRUD operations with objects, instead of complex SQL statements.

To begin, select the Models folder from Solution Explorer. From the Project menu, choose Add Class. Name the class Person. Paste the following properties into the Person class (these properties will become columns in a Person table in SQL Server):

```
public class Person
{
    public int PersonID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

ASP.NET MVC 4 applications require a controller class, which acts as a bridge between the view and the model. You can think of the view as the UI and the model as the data. With a controller added, Visual Studio automatically generates the needed view classes.

Figure 6 Files Added

Controller Folder	PersonController.cs
Models Folder	Person.cs PersonContext.cs
Views/Person Folder	Create.cshtml Delete.cshtml Details.cshtml Edit.cshtml Index.cshtml



**Intense Take-Home
Training for Developers,
Software Architects
and Designers**

**LET US
HEAR
YOU
CODE**



LAS VEGAS | **MARCH**
25-29, 2013
MGM Grand Hotel & Casino

70+ Sessions and Workshops!

Visual Studio 2012 / .NET 4.5
ASP.NET | SharePoint
SQL Server | Windows 8 / WinRT

**Register before
January 30
and save \$400!**

vslive.com/lasvegas Use Promo Code TIP1

FLIP OVER FOR MORE EVENT TOPICS



- ASP.NET
- Azure / Cloud Computing
- Cross-Platform Mobile
- Data Management
- HTML5 / JavaScript
- Deep Dive: SharePoint / Office 365
- Deep Dive: SQL Server
- Windows 8 / WinRT
- WPF / Silverlight
- Visual Studio 2012 / .NET 4.5

vslive.com/lasvegas

Use Promo Code **TIP1**

BONUS LAS VEGAS CONTENT!

DEVELOPER DEEP DIVES
ON SHAREPOINT AND
SQL SERVER – BROUGHT
TO YOU BY:

SharePoint **LIVE!**
TRAINING FOR COLLABORATION

SQL Server **LIVE!**
TRAINING FOR DBAs AND IT PROS



Scan the QR
code for more
information on
Visual Studio Live!

Use Promo Code **TIP1**



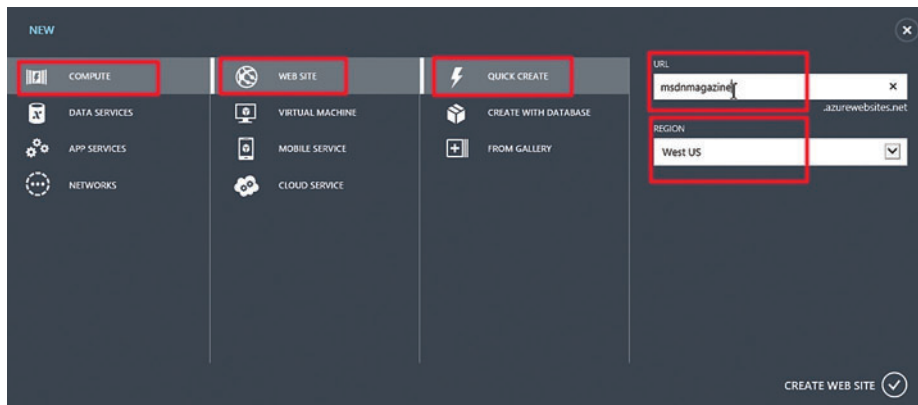


Figure 7 Quick Provisioning of a Windows Azure Web Site Deployment

Before adding the controller class, though, you need to build the solution so that the project adds the appropriate type information needed in future steps. From the Build menu, choose Build Solution. In Solution Explorer you can right-click on the Controllers folder and choose Add | Controller. The Add Controller dialog shown in Figure 5 will appear, and you'll need to set the Scaffolding options with the following values:

1. In the Template dropdown list, select the option MVC Controller with read/write actions and views, using Entity Framework.

You can deploy Web Sites using continuous delivery frameworks.

2. In the Model class dropdown list, select the Person class.
3. In the Data context class list, select <New data context...>. In the dialog box displayed, set the name to Person-Context and click OK.
4. In the Views dropdown list, make sure Razor is selected.
5. Click Add to complete the process.

Adding the controller to the project generated all the necessary Web scaffolding so CRUD operations can be performed against a SQL Server database, resulting in the addition of several files, as shown in Figure 6. If you open the PersonController class, you'll be able to see all the action methods that support Create, Edit, Delete and List operations.

The Next Step—Deployment

As mentioned earlier, you can deploy Web Sites using continuous delivery frameworks such as Git or TFS, or simple tools like FTP or Web Deploy. In this article, we'll use Web Deploy from Visual Studio, but you'll find information for the rest of the deployment methodologies in the Windows Azure Training Kit (bit.ly/Wb0EKZ).

The first step is to log on to the Windows Azure Management Portal. To do so, sign up for a 90-day trial at bit.ly/azuretestdrive. Once you set up

your account, log on with this URL: manage.windowsazure.com. Then click on the Web Sites link on the left side of the browser window. Next, click on NEW | COMPUTE | WEB SITE | QUICK CREATE and type in a desired URL, as shown in Figure 7. We used msdnmagazine, which means our site will be live at <http://msdnmagazine.azurewebsites.net>. You should also select a region.

At this point, your site is almost ready for deploying your content. The provisioning process is extremely quick. Once you see that the status reads "running," you can click on the Web site name and

begin to manage it. There's also a dashboard at the portal that shows CPU time, Requests, Data Out, Data In and HTTP Server Errors.

A key step to deploying your application is to obtain the publish profile file, which contains the settings that Visual Studio will use to perform the deployment. Depending on your browser, you'll be given the ability to download the publish profile file. You should save a copy locally so you can use it later from Visual Studio.

Configuring the Database Server

Before you can go live with your ASP.NET MVC 4 application, you need to configure the database, as shown in Figure 8. As you'll recall, this project is using the Entity Framework Code First paradigm, which means you don't have to worry about all the data management commands typically needed to create and initialize relational data structures. Entity Framework takes care of this.

When you click on ADD, you'll be asked to provide a login name and password, as well as the region where you wish to locate your Windows Azure SQL Database deployment. It's highly recommended you select the same datacenter where your WAWS is running, to minimize latency and avoid bandwidth charges. In our case, the login name is DBAdministrator. This can be important later if you wish to remote in or if you need to build a connection string. Also note that a server name has been provisioned for you (in our case it is ccek4yihqf). You can click on the server name to get more details from the portal (see Figure 9).

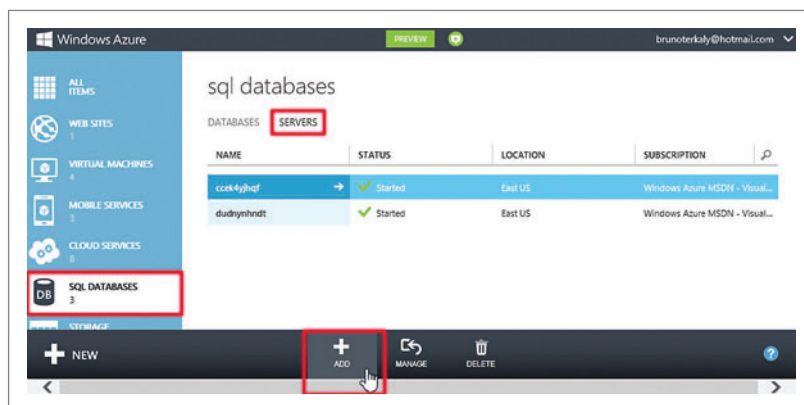


Figure 8 Adding a Server to Host SQL Database

Figure 9 Server Details

Server Name	ccek4yihqf
Server Status	Started
Administrator Login	DBAdministrator
URL	https://ccek4yihqf.database.windows.net

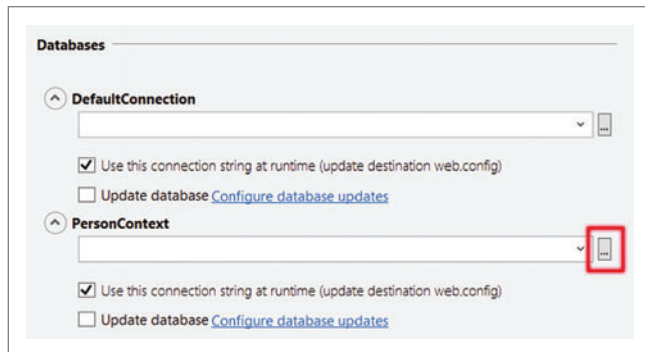


Figure 10 Configuring the Database Connection

An important goal here is to protect your database server information, allowing only specific IP addresses for inbound connections. Click on Configure at the top of the screen. You'll see your current IP address, which you can then use to create a rule and connect to the server directly from that IP address.

Performing a Web Deploy from Visual Studio

We're now ready to finalize this project by deploying it and simultaneously creating a database on the server we just provisioned. Go back to Visual Studio and select View | Solution Explorer. Right-click on MVC-Azure.WebSite and choose Publish. The Publish Web dialog box will appear. The process is wizard-based and the first step is to load the publish profile file into Visual Studio, allowing you to dramatically simplify the deployment process. You'll go through several steps specifying the details of your publishing profile for your ASP.NET MVC 4 application, providing information about how you'd like to deploy your application inside a Microsoft datacenter, including the destination URL and the location of your Windows Azure SQL Database server.

You can specify the Windows Azure SQL Database server your ASP.NET MVC 4 app will use by first clicking Settings on the left side of the dialog box and then clicking on the ellipsis next to PersonContext, as shown in Figure 10.

Now you'll need the information you received from the portal when you created the database server. In our case, the server is tcp:siqxqgihy.database.windows.net, as shown in Figure 11. Note that we prepended

tcp at the front of the server name. You'll also need to enter the administrator name (we entered DBAdministrator in an earlier step) and the password.

Clicking OK physically creates the SQL Database on the server you entered. You'll be asked to confirm this step. Once the database is built, you can choose Publish from the Publish Web Application dialog box. To view exactly what's taking place during the deployment, you can select View | Output window in Visual Studio.

The entire application is now finished, and you'll find our version at msdnmagazine.azurewebsites.net/Person. Keep in mind that we hardly had to write any code at all—it took just a few minutes to accomplish something practical and polished.

What's Next After Deployment?

Once your application has been deployed, it's easy to accomplish tasks such as monitoring, scaling or upgrading your Web site. The first approach is to use the Windows Azure portal, which provides an easy-to-use dashboard that can be accessed from multiple devices. Starting with version 1.8 of the Windows Azure SDK, it's also possible to automate some of these tasks—including management of connection strings and application settings, changing the number of instances or downloading the latest log—by using Windows PowerShell cmdlets or direct Representational State Transfer (REST) API calls.

Wrapping Up

WAWS gives you a way to deploy Web applications almost instantly—with little or no cloud experience. As your requirements grow, you can incorporate other aspects of Windows Azure cloud technologies, such as caching, Service Bus, storage and other value-added services.

Backed by Windows Azure PaaS components, there's no easier way to host scalable, highly available Web applications in the cloud. These features, combined with pre-built frameworks such as WordPress, Drupal, DotNetNuke, and Umbraco, allow developers to focus on building rich Web applications and data repositories, delegating the infrastructure tasks to the Windows Azure platform. ■

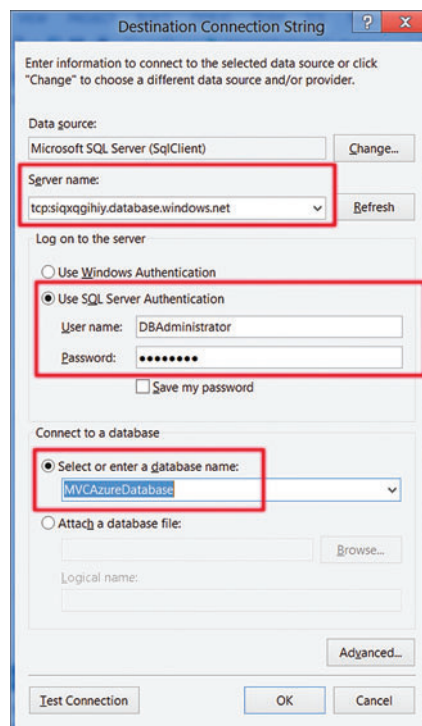


Figure 11 Connecting to the Server and Creating the Database

BRUNO TERKALY is a developer evangelist for Microsoft. His depth of knowledge comes from years of experience in the field, writing code using a multitude of platforms, languages, frameworks, SDKs, libraries and APIs. He spends time writing code, blogging and giving live presentations on building cloud-based applications, specifically using the Windows Azure platform.

RICARDO VILLOBO is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in the supply chain management industry. Holding different technical certifications, as well as a master's degree in business administration from the University of Dallas, he works as a cloud architect in the Windows Azure CSV incubation group for Microsoft.

THANKS to the following technical expert for reviewing this article: Nir Mashkowsky



Are your .NET apps slowing down?

Are your .NET apps slowing down as you increase user activity or transaction load on them? If so then consider using NCache. NCache is an extremely fast and scalable in-memory distributed cache for .NET.

Performance & Scalability thru Data Caching

Cache app data, reduce expensive database trips, and scale your .NET apps.

- Performance: extremely fast in-memory cache
- Linear Scalability: just add servers and keep growing
- 100% uptime: self-healing dynamic cache cluster
- Mirrored, Replicated, Partitioned, and Client Cache topologies

Use for Following in Web Farms

- ASP.NET Session Storage: Replicate sessions for reliability
- ASP.NET View State: Cache it to reduce payload sent to the browser
- ASP.NET Output Cache: Cache page output & improve response time
- NHibernate Level-2 Cache: Plug-in without any code change
- Entity Framework Cache: Plug-in without any code change

Fast Runtime Data Sharing between Apps

- Powerful event notifications for pub/sub data sharing
- Continuous Query and group based events

Download a 60-day FREE trial today!



www.alachisoft.com



1-800-253-8195

Access Online Services with the Windows Runtime and OAuth

Tim Kulp

Once upon a time, back in the Web 1.0 days, Web sites were silos of content to be read and nothing more. As Web 2.0 buzzed into development shops, Web sites became online services with APIs used by developers to mix and match components, data and functionality. Now, mashups allow developers to access rich content libraries without the overhead of housing the data in their server rooms.

With the Windows Runtime (WinRT), you can bring the power of mashups to your next Windows Store app. Whether you're managing data with XMLHttpRequest (XHR) or authenticating to a remote service with the WebAuthenticationBroker class, Windows Library for JavaScript (WinJS) and the Windows Runtime make mashing online services with your app easy.

Contoso Photo Finish

In this article I'm going to build a mashup called Contoso Photo Finish. It lets runners track their miles and post a picture from their

runs. Many runners like to share information such as distance and location of their runs on social networks. Contoso Photo Finish lets users tell their friends about their runs with comments and pictures on Facebook. This app will connect to two different services:

- Windows SkyDrive to retrieve a picture from the run
- Facebook to post the picture for their friends to view

Contoso Photo Finish will combine these services to provide a connected experience for the user. This article assumes you have Visual Studio 2012 open with the JavaScript | Windows Store | Blank App template ready for you to start coding.

Mashup Hurdles: Authorization and Authentication

If an app (Web or Windows Store) wants to post content to Facebook, the first hurdle to overcome is authentication. Facebook needs to know who's connecting to it. When users try to log in to apps, they claim an identity (usually in the form of a username) and a credential (such as a password, security token, biometric device and so on) to prove they should have access to the requested identity. Authentication is the process of validating a user's identity through a credential.

After authenticating the user, the mashup has another challenge: determining what the user can do in the system. Authorization is the process of permitting or denying actions an identity is attempting to perform based on some attribute or security role membership. As an example, in Twitter my identity is not permitted to delete all the tweets of all users. I'm not authorized to perform that action because I'm not a member of the security role with permission to do that. Together, authentication and authorization (A&A) represent the question: User, who are you and what can you do?

Mashups amplify these challenges because the developer building the mashup doesn't have access to where the identities, credentials and security roles are stored (often referred to as a

This article discusses:

- Authorization and authentication
- Using OAuth
- The authentication process using OAuth clients
- Using the WebAuthenticationBroker class
- Retrieving photos
- Using XHR

Technologies discussed:

Windows 8, Windows Runtime, WinJS, OAuth

Code download available at:

archive.msdn.microsoft.com/mag201301Services

credential store). So if I can't verify who's who and what they can do, how can I build a mashup for apps such as Facebook? OAuth to the rescue!

OAuth: Accessing Resources, Not Apps

OAuth addresses the challenge of A&A for mashups. Imagine App X wants to access content from Online Service Y. Instead of the user authenticating to App X, the user authenticates to Online Service Y because the user's credentials are stored in Online Service Y's credential store. The user then permits App X to access specified resources from Online Service Y for a limited time. Permission to access Online Service Y resources is returned to App X as an access token (sometimes referred to as just a "token").

In traditional Web A&A models, two participants work together to determine a user's access: the app and the user (or the server and client). In OAuth a third participant is introduced: the resource server. Resource servers are the third parties who have a resource (such as a photo) stored on the server that a client needs to access. In Contoso Photo Finish, Facebook is a resource server. The resource that Contoso Photo Finish wants to access is the user's status, in order to post a message.

OAuth Clients and Their Authentication Processes

There are two types of clients in OAuth, and which type to use is decided by the client's level of trust. Confidential clients can keep their credentials secure and are meant for highly trusted environments. Examples of confidential clients are server-side Web apps where the client secret can be maintained in a controlled, secure environment. Confidential clients use the Authorization Code process to obtain a security token by providing the client secret to the resource server as a means of authenticating the client.

Public clients can't keep their credentials secure because they run in a hostile environment. Example public clients would be user-agent apps (that is, JavaScript Web apps) or native apps (such as Windows Store apps). Public clients use the "implicit grant" process to obtain a security token because the client secret can't be stored in a secure manner within a hostile environment that's outside the developer's control.

Windows Store apps can be configured to use either implicit grant or authorization code processes for OAuth (you can read more about implicit grant and authorization code processes at bit.ly/yQjyQZ). It's a security best practice to use implicit grant any time an app is out of the developer's control.

In **Figure 1** I examine an implicit grant process where the client starts the conversation by attempting to determine a user's identity with the resource server.

The steps illustrated in **Figure 1** are explained here:

1. The Windows Store app needs to perform some functionality that requires access to the Facebook API.
2. The user connects to the resource server through a URI that includes information about the Windows Store app trying to access the Facebook API. This is usually in the form of an app ID or client ID code. The user provides a username and password to log in to Facebook.

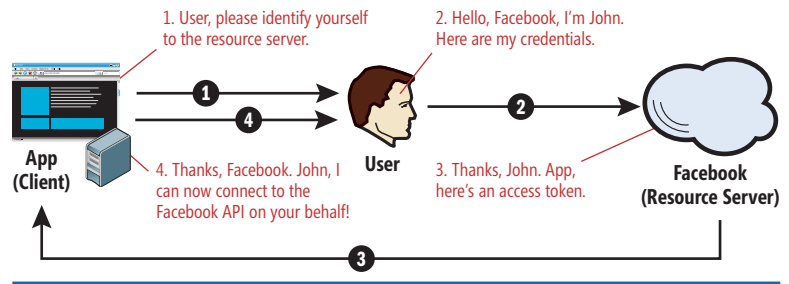


Figure 1 Windows Store App Implicit Grant Conversation

3. Assuming successful login, Facebook provides the Windows Store app with an access token.

4. The Windows Store app can now provide the user with data from the Facebook API using the access token provided by Facebook to get the user's feed, post pictures and so on.

Making this conversation happen is surprisingly easy through the new `Windows.Security.Authentication.Web` namespace.

WebAuthenticationBroker

In Windows Store apps the `WebAuthenticationBroker` class (bit.ly/RW8czT) is the component that will communicate with the resource server, provide the login controls and respond to a successful login, all without needing the Windows Store app to know anything about the user's credentials. For the sample app, Contoso Photo Finish needs to post pictures to Facebook. This requires the user to authenticate to Facebook and receive an access token.

Add a new page control to the Contoso Photo Finish project called `input.html`. By default Visual Studio provides a lot of markup for you. Replace `<p>Content goes here.</p>` in the Main content section with the following button:

```
<input type="button" id="btnAddRun" value="Add Run" />
```

This is the button the user will click to add a run to Photo Finish. Now open `input.js` and add the following function:

```
function btnAddRun_Click(e) {
    var facebookOAuthUrl = "https://www.facebook.com/dialog/oauth";
    var facebookClientId = "[YOUR CLIENT ID]";
    var redirectUrl = "https://www.facebook.com/connect/login_success.html";

    var requestUri = Windows.Foundation.Uri(facebookOAuthUrl +
        "?client_id=" + facebookClientId +
        "&redirect_uri=" + encodeURIComponent(redirectUrl) +
        "&response_type=" +
        "token&scope=read_stream,publish_actions&display=popup");
    var callbackUri = Windows.Foundation.Uri(redirectUrl);

    // Web authentication broker will go here
}
```

This code establishes the variables that will be used in the authentication request. The first variable is the URL of the Facebook OAuth service. Client ID is the app identifier used by Facebook to identify Contoso Photo Finish as the app with which the Facebook API will interact. An identifier such as this is normally assigned to an app when it's registered with the resource server. Some resource servers refer to the identifiers as client id, app id or just id. Which id to use will be found in the resource server's API documentation.

The `redirectUrl` parameter determines where the app will go after the user authenticates and approves the app's access to specified resources. In this case the `redirectUrl` is set to a Facebook standard that's available to all Facebook API apps. Some services require the

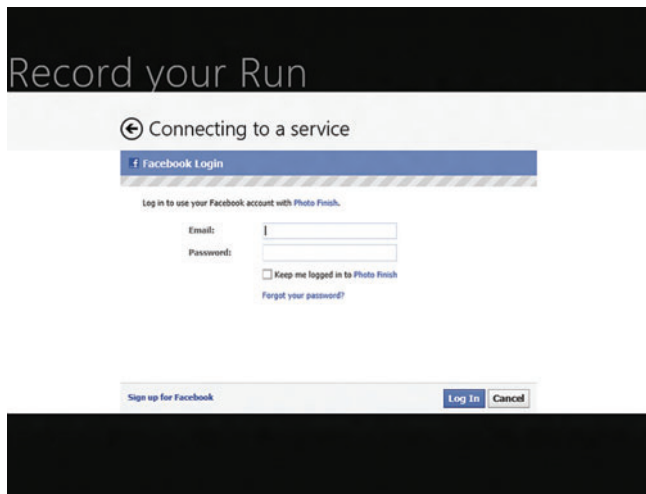


Figure 2 Resource Server's Login Pop-up from `authenticateAsync`

Windows Store app's URI to be identified with the resource server when the app is registered. The app's URI can be found using the Web authentication broker's `getCurrentApplicationCallbackUri` method. This will return the app's local context URI (beginning with "ms-app://"). Some resource servers don't support ms-app:// as a valid protocol for a `redirectUrl`, in which case you should check for a default redirect address such as what Facebook provides.

Next, the `callbackUri` is defined. This is the address that informs the Web authentication broker when authentication is complete and returns control back to the Windows Store app. The broker never actually goes to this URL; it simply watches for the resource server to call for this page and then returns the `callbackUri` with any query string or hash parameters that were appended. In this case, the hash parameter "access_token" will provide Contoso Photo Finish the token needed to interact with the API.

The `WebAuthenticationBroker` class uses the `authenticateAsync` method to connect to and complete the authentication process with the resource server. When `authenticateAsync` is called, the app opens a pop-up that displays the resource server's login screen, as shown in Figure 2. Once authentication is complete or the `callbackUri` is encountered, the pop-up will close.

Figure 3 Working with the Result of the Authentication Process

```
switch (result.responseStatus) {
    case Windows.Security.Authentication.Web.WebAuthenticationStatus.success:
        var fragment = Windows.Foundation.Uri(result.responseData).fragment;
        if (fragment.indexOf("#access_token=") != -1) {
            var token = fragment.substring(
                new String("#access_token=").length,
                fragment.indexOf("&expires_in="));
            // Add API calls here
        }
        break;
    case Windows.Security.Authentication.Web.WebAuthenticationStatus.userCancel:
        Log(window.toStaticHTML(result.responseData));
        Display("User cancelled the authentication to Facebook.");
        break;
    case Windows.Security.Authentication.Web.WebAuthenticationStatus.errorHttp:
        Log(window.toStaticHTML(result.responseData));
        Display("An error occurred while communicating with Facebook.");
        break;
}
```

A key advantage to using this pop-up is that the Windows Store app never handles or needs to know the user's credentials for the resource manager. All the app knows is the access token that's returned by the resource server. This separation keeps the resource server's credentials isolated from the app, which avoids the security risk of the app storing the credentials. On top of the security benefits, the developer doesn't have to code anything to get this interface; it's built in to the `authenticateAsync` method. When developers call the method, the interface comes with it.

Now, back to code. Replace the "Web authentication broker will go here" comment with the following code:

```
Windows.Security.Authentication.Web.WebAuthenticationBroker.
    authenticateAsync(Windows.Security.Authentication.Web.
        WebAuthenticationOptions.none, requestUri, callbackUri)
    .done(
        function (result) {
            // Check the response status here
        },
        function (ex) {
            Log(ex);
        }
    );
```

The `authenticateAsync` method takes three parameters (the third is optional):

1. **WebAuthenticationOptions:** This is used to provide instructions to the Web authentication broker on how to render the authentication dialog or what data to return in the response. In the preceding example, the app uses "none" to show a common implementation that passes no options to the broker using a default setup.
2. **requestUri:** This is the login entry point for the resource server. In this case, Contoso Photo Finish is connecting to Facebook's OAuth service. `requestUri` must be over a secure connection using the HTTPS protocol.
3. **callbackUri:** This is the page that, when navigated to, returns control back to the Web authentication broker as described earlier. This argument is optional, but if the resource server can't (or won't) redirect to ms-app://, this parameter is how the app will escape the resource server's control. For example, in the earlier code when https://www.facebook.com/connect/login_success.html is navigated to after a successful login, the Web authentication broker will take control of the app from the resource server by closing the authentication dialog and processing the success of the promise. `callbackUri` doesn't have to be in the immediate next page; it could be after a wizard or some other process that must occur on the resource server's site. This URI will normally be the same as the `redirectUrl` but provides the flexibility to extend the authentication process, if necessary.

If the Web authentication broker connects to the resource server, the promise succeeds. Detecting the result of the authentication process is done through the `WebAuthenticationResult` object's `ResponseStatus` property. In the preceding code, the result argument is a `WebAuthenticationResult` object with three properties: `Response Data` (data from the resource server), `ResponseErrorDetail` (if something went wrong, what was it?) and `ResponseStatus` (what's the status of the authentication?). Replace the "Check the response status here" comment with the code shown in Figure 3.

Figure 4 Input Fields to Provide Content for the App

```
<p>
  <label>Distance</label>
  <input type="number" min="0" max="15" id="txtDistance"
    required /> miles
</p>
<p>
  <label>Comment</label>
  <input type="text" min="0" max="15" id="txtComment" />
</p>
<p>
  <label>Photo</label>
  <input id="btnSelectPhoto" value="Select Photo" type="button" />
  <img src="" id="imgSelectedPhoto" alt="Selected Photo" />
</p>
<p>
  <input type="button" id="btnAddRun" value="Add Run" />
</p>
```

In the **Figure 3** code, each status is checked, with the Log method recording the information from the resource server and the Display method telling the user what has occurred. For error messages, remember to display user-friendly messages to improve usability and reduce accidental exposure of sensitive information from a system-generated error message. If the authentication is successful, the URI fragment returned from Facebook is parsed and stored in the token variable for use in the API call (see the “Getting and Posting Information via XHR” section for implementation details).

With the btnAddRun_Click function complete, connect it to the btnAddRun object in the WinJS.UI.Pages.define { ready } function:

```
var btnAddRun = document.getElementById("btnAddRun");
if (null != btnAddRun)
  btnAddRun.addEventListener("click", btnAddRun_Click, false);
```

At this point the app has an access token showing that the user is authenticated to the resource server. In the last section the app will execute API commands to send data, but first the app needs something to send to Facebook.

Getting the Picture

Windows 8 provides a variety of contracts that allow apps to talk with each other, such as search, share and file picker. These contracts can turn any app into a mashup with just a few lines of code. Contoso Photo Finish is going to tap into the power of the file picker contract to find an image for the user's run.

Figure 5 Picking a File

```
filePicker.pickSingleFileAsync().then(
  function (storageFile) {
    if (storageFile) {
      selectedPhotoFile = storageFile;
      selectedPhotoFile.openAsync(
        Windows.Storage.FileAccessMode.read).then(
        function (stream) {
          selectedPhotoStream = stream;
          document.getElementById("imgSelectedPhoto").src =
            URL.createObjectURL(selectedPhotoFile);
        },
        function (ex) {
          Log(ex);
          Display("An error has occurred while reading the file.");
        });
    }
    else {
      Display("File was not selected");
    }
  });
```

One of the many things I love about my Windows Phone is the integration with SkyDrive. I can upload my photos instantly to storage in the cloud so the next time I break my phone (which is often), my pictures are waiting for me online. The SkyDrive app for Windows 8 provides data to the file picker, making the selection of files from my SkyDrive account as simple as selecting them from the picture library. The next part of the Contoso Photo Finish mashup will consume data from the SkyDrive app through the file picker. To do this, input.html needs some, well . . . inputs.

Replace the btnAddRun button with the code shown in **Figure 4**. This code includes the input fields for the user to provide content for Contoso Photo Finish. The btnSelectPhoto button will use the file picker to select which file on the system to use. Add a new function to input.js that will be the click handler for btnSelectPhoto:

```
function btnSelectPhoto_Click(e) {
  var imgSelectedPhoto = document.getElementById("imgSelectedPhoto");

  var filePicker = new Windows.Storage.Pickers.FileOpenPicker();
  filePicker.fileTypeFilter.replaceAll([".jpg", ".jpeg", ".png"]);
  filePicker.suggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
  filePicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;

  // Pick file here
}
```

The function begins by setting up the imgSelectedPhoto variable, which will be used to display the selected photo to the user. Next, the code creates a file picker object. The file picker object allows Contoso Photo Finish to choose files or folders (in this case just files) on the system or in other apps participating in the file picker contract to open and interact within the app. By using the file type filter, the code limits which file extensions are accessible to the file picker. The app can only load images to Facebook (by design), so limiting the file picker to only work with specified image file types keeps the user from selecting files that have invalid extensions that don't meet the desired functionality. Also, due to the app's focus on images, the file picker's start location is set to the picture library. This could be a variety of default locations (such as music library, document library, home group and so on), but when dealing with pictures, the picture library is a common-sense starting point. The last setting for the file picker is to set the viewMode to thumbnail. This displays a preview of the file and is ideal for image selection.

With the options set, it's time to select the file to use for the run. First, add these two variable declarations directly below the “use strict” statement:

```
var selectedPhotoStream = null;
var selectedPhotoFile = null;
```

These will hold the file and stream values for the btnAddRun_Click function when the data is loaded to Facebook. Now replace the “Pick file here” comment with the code shown in **Figure 5**.

Figure 5 looks like a lot of code, but it boils down to three actions:

1. Select the file that the app will be using via the file picker (pickSingleFileAsync).
2. Open the file stream to be read (openAsync).
3. Store the stream and file into variables for later use.

All of the code is standard for working with files and streams with one exception: URL.createObjectURL takes an object and builds a URL for the object to be displayed through an image object. The createObjectURL method works with many object types including

Stream, StorageItem and MediaCapture. In general it's used to display media content (image, audio or video) in a Windows Store app. One thing to keep in mind when using createObjectURL is that when you're finished using the URL, make sure to dispose of it through the URL.revokeObjectURL method. This will ensure optimal memory usage and prevent the Windows Store app from getting bogged down with too many temporary URLs. For more information about createObjectURL, check out the MSDN documentation at bit.ly/Xdhz0m.

Finally, associate the btnSelectPhoto_Click event with the btnSelectPhoto object. Add the following code in the WinJS.UI.Pages.define { ready } function:

```
var btnSelectPhoto = document.getElementById("btnSelectPhoto");
if (null != btnSelectPhoto)
    btnSelectPhoto.addEventListener(
        "click", btnSelectPhoto_Click, false);
```

At this point, Contoso Photo Finish has content to post and I have the mechanism to authenticate to Facebook for posting. Now the app just needs to interact with the API and throw the content online.

Getting and Posting Information via XHR

Remember when AJAX was new and exciting? XHR came on to the scene in Internet Explorer 5.5 and made Web developers start to rethink how Web apps were being made. Time passed and AJAX grew with many different libraries (such as jQuery) into an easy-to-understand and (more important) easy-to-implement solution. WinJS.xhr continues this tradition with a simple API to get and post data to online services.

Return to the btnAddRun_Click function and replace the "Add API calls here" comment with the code shown in **Figure 6**.

Earlier, the app stored the StorageFile and Stream into the selectedPhotoFile and selectedPhotoStream variables. MSApp.createBlobFromRandomAccessStream takes the selectedPhotoStream and generates a Blob object (bit.ly/Stfu9z) that the app will later pass to Facebook as a POST parameter. This is a useful feature for converting WinRT objects into a format that can be transferred via HTTP.

Next, the code uses the FormData object (new to HTML5) to create the parameters that will be sent in the HTTP POST.

Figure 6 Generating a Blob Object from selectedPhotoStream

```
var fileBlob = MSApp.createBlobFromRandomAccessStream(
    selectedPhotoFile.contentType,
    selectedPhotoStream);

var message = "I just ran " + document.getElementById(
    "txtDistance").value + " miles with PhotoFinish! " +
    document.getElementById("txtComment").value;
var data = new FormData();
data.append("source", fileBlob);
data.append("filename", selectedPhotoFile.name);
data.append("access_token", token);
data.append("message", window.toStaticHTML(message));

WinJS.xhr({
    type: "POST",
    url: "https://graph.facebook.com/me/photos",
    data: data,
}).then(
    function (photoid_response) {
        ProcessResponse(photoid_response);
    },
    function (ex) {
        Display("An error occurred while posting the photo.");
        Log(ex);
    });
```

FormData is a key/value pair object with just one method: append. Using the append method, developers can build form fields dynamically and submit them with a POST as if the form's submit event was called. FormData also provides the parameters as if they belonged to a form with multipart/form-data encoding. This allows developers to post any input type (including files) to the target server.

Notice in the FormData.append method call that I use toStaticHTML (bit.ly/ZRKbka) to ensure that the message content is safe for delivery. Using toStaticHTML will remove any event attributes or script content from the message variable prior to adding it to the FormData object. While I imagine Facebook is good about preventing cross-site scripting attacks (among others), as a mashup developer, I want to provide clean content to my fellow apps. The Internet is a big place, so we all need to watch out for each other.

The rest of the code block is the WinJS.xhr call. This time around, the code uses some more attributes of XHR, including:

- **type:** This sets the HTTP method to be used. By default type is set to GET. This code uses POST because the app is sending content to the Facebook API.
- **data:** This consists of parameters to pass with the POST.

When the promise is returned in the success method, Contoso Photo Finish processes the photo ID for later retrieval. If an error occurs, the standard error message is displayed and the exception is logged.

WinJS.xhr is a lot like other XHR wrappers. If you're familiar with JavaScript libraries such as jQuery, you'll be able to pick up WinJS.xhr easily. One gotcha that you could encounter is that, unlike XHR, there's no timeout option on the WinJS.xhr method. Setting a timeout is accomplished by wrapping the WinJS.xhr call with a WinJS.Promise.timeout method (bit.ly/Qgtx7a). By adding the following code to the beginning of the WinJS.xhr call, I set a timeout of 10 seconds to the POST:

```
WinJS.Promise.timeout(1000, WinJS.xhr({ ... }));
```

If the WinJS.xhr promise doesn't complete within 10 seconds, the promise will timeout and be handled through the timeout promise's error function.

First Steps in Mashing Your App

In this article I examined authenticating, getting files and sending data with WinJS and the Windows Runtime. These foundation skills can be built upon to design Windows Store apps that are limited only by your imagination and developer keys. Take the material in this article and explore your favorite online service. Using WinJS.xhr, your Windows Store app can interact with the countless APIs available online. With the Web authentication broker, your app can connect users with their online personalities, content and communities using OAuth or OpenID. WinJS and the Windows Runtime give you all the tools to easily build an app that's greater than the sum of its online services. ■

TIM KULP leads the development team at FrontierMEDEX in Baltimore. You can find Kulp on his blog at seccode.blogspot.com or on Twitter at [Twitter.com/seccode](https://twitter.com/seccode), where he talks code, security and the Baltimore foodie scene.

THANKS to the following technical experts for reviewing this article:
Sunil Gottumukkala and Jeremy Viegas

sponsored by Microsoft

ALM Summit 3

expertise worth sharing

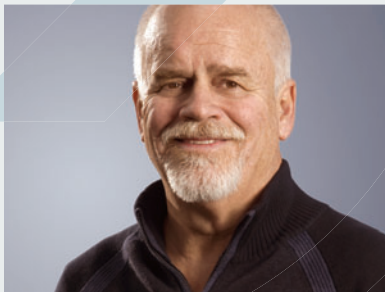
January 29-31, 2013 - Microsoft Conference Center, Redmond

Register at www.alm-summit.com

follow us on



keynote speakers



Dean Leffingwell

Creator, Scaled Agile Framework

"Being Agile. Scaling Up. Staying Lean"



Scott Porad

CTO, Cheezburger Network

"Lean Startups, Cheezburgers, and You"



Brian Harry

Technical Fellow, Microsoft

"Building an Engineering Organization for Continuous Delivery"



Jason Zander

Corporate Vice President, Microsoft

"Continuous Services and Connected Devices"



Sam Guckenheimer

Product Owner, Microsoft

"Reimagining the Application Lifecycle"



James Whittaker

Testing Guru, Microsoft

"Re-inventing the Internet"

conference tracks

ALM Leadership • Agile Development • DevOps • Testing

sponsors



diamond



Improving the Profession of Software Development
platinum



where technology meets teamwork





**Intense Take-Home
Training for Developers,
Software Architects
and Designers**

LET US HEAR YOU CODE





LAS VEGAS | **MARCH 25-29, 2013**
MGM Grand Hotel & Casino



SUPER EARLY BIRD SPECIAL

**Register before
January 30 and
save \$400!**

Use Promo Code LVJAN

Everyone knows all the *really* cool stuff happens behind the scenes. Get an all-access look at the Microsoft Platform and practical, unbiased, developer training at Visual Studio Live! Las Vegas.

Topics will include:

- ASP.NET
- Azure / Cloud Computing
- Cross-Platform Mobile
- Data Management
- HTML5 / JavaScript
- SharePoint / Office 365
- SQL Server
- Windows 8 / WinRT
- WPF / Silverlight
- Visual Studio 2012 / .NET 4.5



TURN THE PAGE FOR MORE EVENT DETAILS

vslive.com/lasvegas

CODE WITH .NET ROCKSTARS AND LEARN HOW TO MAXIMIZE THE DEVELOPMENT CAPABILITIES OF VISUAL STUDIO AND .NET DURING 5 ACTION-PACKED DAYS OF PRE- AND POST-CONFERENCE WORKSHOPS, 70+ SESSIONS LED BY EXPERT INSTRUCTORS AND KEYNOTES BY INDUSTRY HEAVYWEIGHTS.



BONUS LAS VEGAS CONTENT!

DEVELOPER DEEP DIVES
ON SHAREPOINT AND
SQL SERVER – BROUGHT
TO YOU BY:

SharePoint **LIVE!**
TRAINING FOR COLLABORATION

SQL Server **LIVE!**
TRAINING FOR DBAS AND IT PROS

TRACKS & SESSIONS

ASP.NET

Sessions Include:

- 25 Tips and Tricks for the ASP.NET Developer
- From 0 to Web Site in 60 Minutes with Web Matrix
- Building Single Page Web Applications with HTML5, ASP.NET MVC4 and Web API

Azure / Cloud Computing

Sessions Include:

- Bringing Open Source to Windows Azure: A Match Made in Heaven
- Elevating Windows Azure Deployments
- Workshop: Services – Using WCF and ASP.NET Web API

Cross-Platform Mobile

Sessions Include:

- Sharing up to 80% of code building Mobile apps for iOS, Android, WP 8 and Windows 8
- iOS Development Survival Guide for the .NET Guy
- Building Multi-Platform Mobile Apps with Push Notifications

Data Management

Sessions Include:

- Busy Developer's Guide to MongoDB
- Busy Developer's Guide to Cassandra
- LINQ performance and Scalability

HTML5 / JavaScript

Sessions Include:

- jQuery Fundamentals
- Tips for building Multi-Touch Enabled Web Sites
- Build Speedy Azure Applications with HTML 5 and Web Sockets Today

SharePoint **LIVE!**

TRAINING FOR COLLABORATION

DEEP DIVE:

SharePoint / Office 365

Sessions Include:

- Building Your First SharePoint 2013 Application Using Visual Studio 2012
- Getting Started with Microsoft Office 365 SharePoint Online Development
- Workshop: SharePoint 2013 Developer Boot Camp

SQL Server **LIVE!**

TRAINING FOR DBAS AND IT PROS

DEEP DIVE: SQL Server

Sessions Include:

- SQL Server Data Tools
- Big Data-BI Fusion: Microsoft HDInsight & MS BI
- Workshop: SQL Server 2012

Visual Studio 2012 / .NET 4.5

Sessions Include:

- Mastering Visual Studio 2012
- Design for Testability: Mocks, Stubs, Refactoring, and User Interfaces
- Workshop: ALM with Visual Studio 2012 and Team Foundation Server 2012

TWindows 8 / WinRT

Sessions Include:

- New XAML controls in Windows 8
- Cross Win8/WP8 Apps
- Workshop: Build a Windows 8 Application in a Day

WPF / Silverlight

Sessions Include:

- Building Extensible XAML Client Apps
- Migrating from WPF or Silverlight to WinR



LAS VEGAS | **MARCH 25-29, 2013**

THE REVIEWS ARE IN! HERE'S WHAT SOME PAST ATTENDEES HAD TO SAY ABOUT VISUAL STUDIO LIVE! . . .

"The Visual Studio Live! event is an electrifying moment for a software developer. It's one of the few places where I can bond with other developers, share memorable moments, and just have an amazing time doing what we love — coding. The event has definitely surpassed my expectations."

Tim Moore, Software Engineer, Civil Air Patrol

"Well organized, lots of sessions to choose from and a wonderful location. Money well spent!"

Rakesh R. Naidu, Software Developer, USAC

"The quality of presenters has been exceptional."

Matthew Hubbell, Developer, DieBold, Inc.

"Since leaving the industry in 2003 to join the ranks of academia it has been a struggle to maintain contact with the pulse of platform and tool evolution. VSLive is an ideal way to be totally immersed in an environment that provides a 'rapid' recharge of all the latest news & technologies from a user-centric viewpoint. I get more in one week at VSLive than I can get from a full three months of research over summer break."

Robert Logan, Assistant Professor, Kent State University

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive – @VSLive



facebook.com – Search "VSLive"



linkedin.com – Join the "VSLive" group!



Scan the QR code to register or for more event details.

Register at vslive.com/lasvegas

Use Promo Code LVJAN

TypeScript: Making .NET Developers Comfortable with JavaScript

Shayne Boyer

No doubt you have a huge investment in the Microsoft .NET Framework, and it's a truly substantial platform with abundant tools available. If you combine your knowledge of C# or Visual Basic .NET with XAML, the market for your existing skills might seem almost limitless. Today, however, you need to consider a language that's been well-established for some time and has, in the last few years, really taken off as the application platform caught up. I'm talking about JavaScript, of course. The growth and capability of JavaScript applications is huge. Node.js, an entire platform for developing scalable JavaScript applications, has become enormously popular, and it's even deployable on Windows Azure. Moreover, JavaScript can be used with HTML5 for game development, mobile applications and even Windows Store apps.

As a .NET developer, you can't ignore the capabilities of JavaScript, nor its spread in the marketplace. When I make this statement to colleagues, I often hear groans about how JavaScript is hard to work with, there's no strong typing, no class structures. I combat such arguments by responding that JavaScript is a functional language and there are patterns for accomplishing what you want.

This is where TypeScript comes in. TypeScript isn't a new language. It's a superset of JavaScript—a powerful, typed superset, which means that all JavaScript is valid TypeScript, and what is produced by the

compiler is JavaScript. TypeScript is an open source project, and all of the information related to the project can be found at typescriptlang.org. At the time of this writing, TypeScript was in preview version 0.8.1.

In this article, I'll cover the basic concepts of TypeScript in the form of classes, modules and types, to show how a .NET developer can become more comfortable with a JavaScript project.

Classes

If you work with languages such as C# or Visual Basic .NET, classes are a familiar concept to you. In JavaScript, classes and inheritance are accomplished through patterns such as closures and prototypes. TypeScript introduces the classical type syntax you're used to and the compiler produces the JavaScript that accomplishes the intent. Take the following JavaScript snippet:

```
var car;
car.wheels = 4;
car.doors = 4;
```

This seems simple and straightforward. However, .NET developers have been hesitant to really get into JavaScript due to its loose approach to object definition. The car object can have additional properties added later without enforcement and without knowing what data type each represents, and thus throw exceptions during runtime. How does the TypeScript class model definition change this, and how do we inherit and extend car? Consider the example in **Figure 1**.

Figure 1 Objects in TypeScript and JavaScript

TypeScript	JavaScript
<pre>class Auto{ wheels; doors; }</pre>	<pre>var Auto = (function () { function Auto() { } return Auto; })(); var car = new Auto(); car.wheels = 2; car.doors = 4;</pre>
<pre>var car = new Auto(); car.wheels = 2; car.doors = 4;</pre>	

At the time of this writing TypeScript was in preview version 0.8.1. All information is subject to change.

This article discusses:

- Classes, modules and types
- Using existing code and libraries
- Windows 8 and TypeScript

Technologies discussed:

TypeScript, JavaScript, Windows 8

Figure 2 A Simple Constructor

TypeScript	JavaScript
<pre>class Auto{ wheels; doors; constructor(w, d){ this.wheels = w; this.doors = d; } } var car = new Auto(2, 4);</pre>	<pre>var Auto = (function () { function Auto(w, d) { this.wheels = w; this.doors = d; } return Auto; })(); var car = new Auto(2, 4);</pre>

Figure 3 A Simple Constructor, Modified

TypeScript	JavaScript
<pre>class Auto{ wheels; doors; constructor(w = 4, d?){ this.wheels = w; this.doors = d; } } var car = new Auto();</pre>	<pre>var Auto = (function () { function Auto(w, d) { this.wheels = w; this.doors = d; } return Auto; })(); var car = new Auto(4, 2);</pre>

On the left is a nicely defined class object called car, with the properties wheels and doors. On the right, the JavaScript produced by the TypeScript compiler is almost the same. The only difference is the Auto variable.

In the TypeScript editor, you can't add an additional property without getting a warning. You can't simply start by using a statement such as `car.trunk = 1`. The compiler will complain, "No trunk property exists on Auto," which is a godsend to anyone who has ever had to track down this gotcha because of the flexibility of JavaScript—or, depending on your perspective, the "laziness" of JavaScript.

Constructors, though available in JavaScript, are enhanced with the TypeScript tooling again by enforcing the creation of the object during compile time, and not allowing the object to be created without passing in the proper elements and types in the call.

Not only can you add the constructor to the class, but you can make the parameters optional, set a default value or shortcut the property declaration. Let's look at three examples that show just how powerful TypeScript can be.

Figure 2 shows the first example, a simple constructor in which the class is initialized by passing in the wheels and doors parameters (represented here by `w` and `d`). The produced JavaScript (on

Figure 4 The Auto Declaration Feature

TypeScript	JavaScript
<pre>class Auto{ constructor(public wheels = 4, public doors?){ } } var car = new Auto(); car.doors = 2;</pre>	<pre>var Auto = (function () { function Auto(wheels, doors) { if (typeof wheels === "undefined") { wheels = 4; } this.wheels = wheels; this.doors = doors; } return Auto; })(); var car = new Auto(); car.doors = 2;</pre>

Figure 5 Adding the Motorcycle Class

<pre>class Auto{ constructor(public mph = 0, public wheels = 4, public doors?){ } drive(speed){ this.mph += speed; } stop(){ this.mph = 0; } } class Motorcycle extends Auto { doors = 0; wheels = 2; } var bike = new Motorcycle();</pre>

the right) is almost equivalent, but as the dynamics and needs of your application expand, that won't always be the case.

In **Figure 3**, I've modified the code in **Figure 2**, defaulting the wheels parameter (`w`) to 4 and making the doors parameter (`d`) optional by inserting a question mark to the right of it. Notice, as in the previous example, that the pattern of setting the instance property to the arguments is a common practice that uses the "this" keyword.

Here's a feature I'd love to see in the .NET languages: being able to simply add the public keyword before the parameter name in the constructor to declare the property on the class. The private keyword is available and accomplishes the same auto declaration, but hides the property of the class.

Default values, optional parameters and type annotations are extended with the TypeScript auto property declaration feature,

Figure 6 The Compiler-Produced JavaScript

<pre>var __extends = this.__extends function (d, b) { function __() { this.constructor = d; } __.prototype = b.prototype; d.prototype = new __(); } var Auto = (function () { function Auto(mph, wheels, doors) { if (typeof mph === "undefined") { mph = 0; } if (typeof wheels === "undefined") { wheels = 4; } this.mph = mph; this.wheels = wheels; this.doors = doors; } Auto.prototype.drive = function (speed) { this.mph += speed; }; Auto.prototype.stop = function () { this.mph = 0; }; return Auto; })(); var Motorcycle = (function (_super) { __extends(Motorcycle, _super); function Motorcycle() { _super.apply(this, arguments); this.doors = 0; this.wheels = 2; } return Motorcycle; })(Auto); var bike = new Motorcycle();</pre>

Figure 7 Wrapping the Auto Class in a Module

```
module Example {
  class Auto{

    constructor(public mph : number = 0,
      public wheels = 4,
      public doors?){
    }

    drive(speed){
      this.mph += speed;
    }
    stop(){
      this.mph = 0;
    }
  }

  export class Motorcycle extends Auto
  {
    doors = 0;
    wheels = 2;
  }
}
var bike = new Example.Motorcycle();
```

making it a nice shortcut—and making you more productive. Compare the script in **Figure 4**, and you can see the differences in complexity start to surface.

Classes in TypeScript also provide inheritance. Staying with the Auto example, you can create a Motorcycle class that extends the initial class. In **Figure 5**, I also add drive and stop functions to the base class. Adding the Motorcycle class—which inherits from Auto and sets the appropriate properties for doors and wheels—is accomplished with a few lines of code in TypeScript.

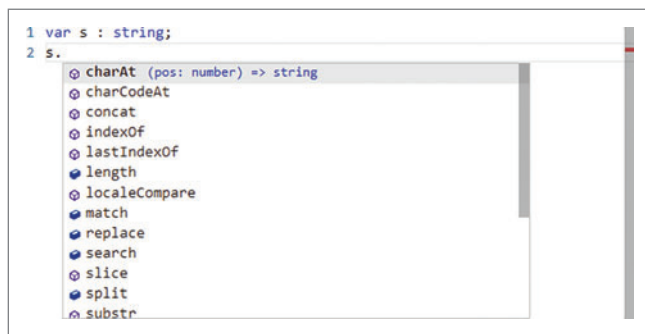


Figure 8 An Example of TypeScript IntelliSense

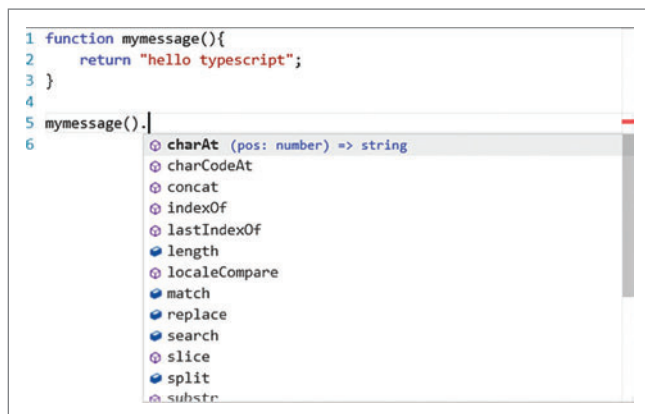


Figure 9 An Example of Type Inference

Figure 10 The Trip Interface

```
interface Trip{
  destination : string;
  when: any;
}

class Auto{
  wheels : number;
  doors : number;

  travel(t : Trip) {
    //..
  }
}

var car = new Auto();
car.doors = 4;
car.wheels = 4;

car.travel({destination: "anywhere", when: "now"});
```

One important thing to mention here is that, at the top of the compiler-produced JavaScript, you'll see a small function called “___extends,” as shown in **Figure 6**, which is the only code ever injected into the resulting JavaScript. This is a helper class that assists in the inheritance functionality. As a side note, this helper function has the exact same signature regardless of the source, so if you're organizing your JavaScript in multiple files and use a utility such as SquishIt or Web Essentials to combine your scripts, you might get an error depending on how the utility rectifies duplicated functions.

Modules

Modules in TypeScript are the equivalent of namespaces in the .NET Framework. They're a great way to organize your code and to encapsulate business rules and processes that would not be possible without this functionality (JavaScript doesn't have a built-in way to provide this function). The module pattern, or dynamic name-spacing, as in JQuery, is the most common pattern for namespaces in JavaScript. TypeScript modules simplify the syntax and produce the same effect. In the Auto example, you can wrap the code in a module and expose only the Motorcycle class, as shown in **Figure 7**.

The Example module encapsulates the base class, and the Motorcycle class is exposed by prefixing it with the export keyword.

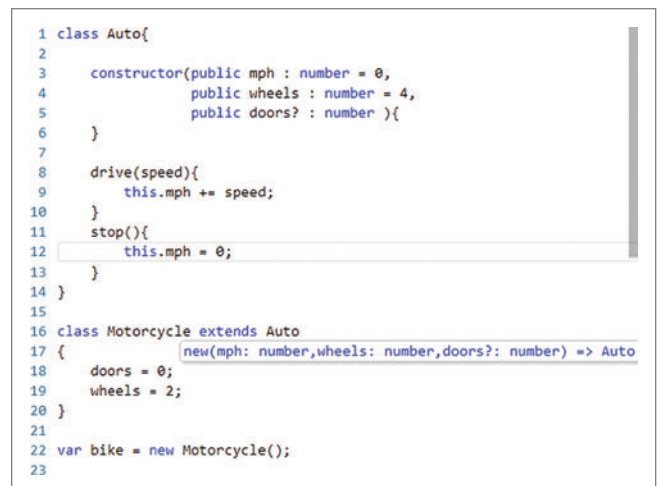


Figure 11 Annotations Assist in Maintaining Your Code

This allows an instance of Motorcycle to be created and all of its methods to be used, but the Auto base class is hidden.

Another nice benefit of modules is that you can merge them. If you create another module also named Example, TypeScript assumes that the code in the first module and the code in new module are both accessible through Example statements, just as in namespaces.

Modules facilitate the maintainability and organization of your code. With them, sustaining large-scale applications becomes less of a burden on development teams.

Types

The lack of type safety is one of the louder complaints I've heard from developers who don't swim in the JavaScript pool every day. But type safety is available in TypeScript (that's why it's called TypeScript) and it goes beyond just declaring a variable as a string or a Boolean.

In JavaScript, the practice of assigning foo to x and then later in the code assigning 11 to x is perfectly acceptable, but it can drive you mad when you're trying to figure out why you're getting the ever-present NaN during runtime.

The type safety feature is one of the biggest advantages of TypeScript, and there are four inherent types: string, number, bool and any. **Figure 8** shows the syntax for declaring the type of the variable s and the IntelliSense that the compiler provides once it knows what actions you can perform based on the type.

Beyond allowing the typing of a variable or function, TypeScript has the ability to infer types. You can create a function that simply returns a string. Knowing that, the compiler and tools provide type inference and automatically show the operations that can be performed on the return, as you can see in **Figure 9**.

The benefit here is that you see that the return is a string without having to guess. Type inference is a major help when it comes to working with other libraries that developers reference in their code, such as JQuery or even the Document Object Model (DOM).

The other way to take advantage of the type system is through annotations. Looking back, the original Auto class was declared with just wheels and doors. Now, through annotations, we can ensure that the proper types are set when creating the instance of Auto in car:

```
class Auto{
  wheels : number;
  doors : number;
}

var car = new Auto();
car.doors = 4;
car.wheels = 4;
```

However, in the JavaScript that's produced, the annotations are compiled away, so there's no fat and no additional dependencies to worry about. The benefit again is strong typing and, additionally, eliminating the simple errors that are generally found during runtime.

Interfaces provide another example of the type safety offered in TypeScript. Interfaces allow you to define the shape of an object. In **Figure 10**, a new method named travel has been added to the Auto class and it accepts a parameter with a type of Trip.

If you try to call the travel method with anything other than the correct structure, the design-time compiler gives you an error. In comparison, if you entered this code in JavaScript, say into a .js file, most likely you wouldn't catch an error like this until you ran the application.

In **Figure 11**, you can see that leveraging type annotations strongly assists not only the initial developer but also any subsequent developer who has to maintain the source.

Figure 12 Creating a Declaration File

TypeScript	JavaScript
<pre>function gradeAverage(grades : string[]) { var total = 0; var g = null; var i = -1; for(i = 0; i < grades.length; i++) { g = grades[i]; total += getPointEquiv(grades[i]); } var avg = total / grades.length; return getLetterGrade(Math.round(avg)); } function getPointEquiv(grade : string) { var res; switch(grade) { case "A": { res = 4; break; } case "B": { res = 3; break; } case "C": { res = 2; break; } case "D": { res = 1; break; } case "F": { res = 0; break; } } return res; } function getLetterGrade(score : number) { if(score < 1) { return "F"; } if(score > 3) { return "A"; } if(score > 2 && score < 4) { return "B"; } if(score >= 1 && score <= 2) { return "C"; } if(score > 0 && score < 2) { return "D"; } }</pre>	<pre>function gradeAverage(grades) { var total = 0; var g = null; var i = -1; for(i = 0; i < grades.length; i++) { g = grades[i]; total += getPointEquiv(grades[i]); } var avg = total / grades.length; return getLetterGrade(Math.round(avg)); } function getPointEquiv(grade) { var res; switch(grade) { case "A": { res = 4; break; } case "B": { res = 3; break; } case "C": { res = 2; break; } case "D": { res = 1; break; } case "F": { res = 0; break; } } return res; } function getLetterGrade(score) { if(score < 1) { return "F"; } if(score > 3) { return "A"; } if(score > 2 && score < 4) { return "B"; } if(score >= 1 && score <= 2) { return "C"; } if(score > 0 && score < 2) { return "D"; } }</pre>

Existing Code and Libraries

So what about your existing JavaScript code, or what if you love building on top of Node.js or use libraries such as toastr, Knockout or JQuery? TypeScript has declaration files to help. First, remember that all JavaScript is valid TypeScript. So if you have something homegrown, you can copy that code right into the designer and the compiler will produce the JavaScript one for one. The better option is to create your own declaration file.

For the major libraries and frameworks, a gentleman by the name of Boris Yankov (twitter.com/borisyankov on Twitter) has created a nice repository on GitHub (github.com/borisyankov/DefinitelyTyped) that has a number of declaration files for some of the most popular JavaScript libraries. This is exactly what the TypeScript team hoped would happen. By the way, the Node.js declaration file was created by the TypeScript team and is available as a part of the source code.

Creating a Declaration File

If you can't locate the declaration file for your library or if you're working with your own code, you'll need to create a declaration file. You start by copying the JavaScript into the TypeScript side and adding the type definitions, and then use the command-line tool to generate the definition file (*.d.ts) to reference.

Figure 12 shows a simple script for calculating grade point average in JavaScript. I copied the script into the left side of the editor and added the annotations for the types, and I'll save the file with the .ts extension.

Next, I'll open a command prompt and use the TypeScript command-line tool to create the definition file and resulting JavaScript:

```
tsc c:\gradeAverage.ts -declarations
```

Figure 13 Declaration Files for Windows 8

```
/// <reference path="winjs.d.ts" />
/// <reference path="winrt.d.ts" />
/// <reference path="jquery.d.ts" />

module Data {

    class Location {
        longitude: any;
        latitude: any;
        url: string;
        retrieved: string;
    }

    var locator = new Windows.Devices.Geolocation.Geolocator();
    locator.getGeopositionAsync().then(function (pos) {

        var myLoc = new Location();

        myLoc.latitude = pos.coordinate.latitude;
        myLoc.longitude = pos.coordinate.longitude;
        myLoc.retrieved = Date.now.toString();
        myLoc.url = "http://dev.virtualearth.net/REST/v1/Imagery/Map/Road/" +
            + myLoc.latitude + "," + myLoc.longitude
            + "15?mapSize=500,500&pp=47.620495,-122.34931;21;AA&pp="
            + myLoc.latitude + "," + myLoc.longitude
            + ";;AB&pp=" + myLoc.latitude + "," + myLoc.longitude
            + ";22&key=BingMapsKey";

        var img = document.createElement("img");
        img.setAttribute("src", myLoc.url);
        img.setAttribute("style", "height:500px;width:500px;");

        var p = $("p");
        p.append(img);
    });
}
```

The compiler creates two files: gradeAverage.d.ts is the declaration file and gradeAverage.js is the JavaScript file. In any future TypeScript files that need the gradeAverage functionality, I simply add a reference at the top of the editor like this:

```
/// <reference path="gradeAverage.d.ts">
```

Then all the typing and tooling is highlighted when referencing this library, and that's the case with any of the major libraries you may find at the DefinitelyTyped GitHub repository.

A great feature the compiler brings in declaration files is the ability to auto-traverse the references. What this means is if you reference a declaration file for jQueryUI, which in turn references jQuery, your current TypeScript file will get the benefit of statement completion and will see the function signatures and types just as if you had referenced jQuery directly. You can also create a single declaration file—say, "myRef.d.ts"—that contains the references to all the libraries you intend to use in your solution, and then make just a single reference in any of your TypeScript code.

Windows 8 and TypeScript

With HTML5 a first-class citizen in the development of Windows Store apps, developers have wondered whether TypeScript can be used with these types of apps. The short answer is yes, but some setup is needed in order to do so. At the time of this writing, the tooling available either through the Visual Studio Installer or other extensions hasn't completely enabled the templates within the JavaScript Windows Store app templates in Visual Studio 2012.

There are three key declaration files available in the source code at typescript.codeplex.com—winjs.d.ts, winrt.d.ts and lib.d.ts. Referencing these files will give you access to the WinJS and WinRT JavaScript libraries that are used in this environment for accessing the camera, system resources and so forth. You may also add references to jQuery to get the IntelliSense and type safety features I've mentioned in this article.

Figure 13 is a quick example that shows the use of these libraries to access a user's geolocation and populate a Location class. The code then creates an HTML image tag and adds a static map from the Bing Map API.

Wrapping Up

The features TypeScript adds to JavaScript development are small, but yield large benefits to .NET developers who are accustomed to similar features in the languages they use for regular Windows application development.

TypeScript is not a silver bullet, and it's not intended to be. But for anyone who's hesitant to jump into JavaScript, TypeScript is a great language that can ease the journey. ■

SHAYNE BOYER is a Telerik MVP, Nokia Developer Champion, MCP, INETA speaker and a solutions architect in Orlando, Fla. He has been developing Microsoft-based solutions for the past 15 years. Over the past 10 years, he has worked on large-scale Web applications, with a focus on productivity and performance. In his spare time, Boyer runs the Orlando Windows Phone and Windows 8 User Group, and blogs about the latest technology at tattoocoder.com.

THANKS to the following technical expert for reviewing this article:
Christopher Bennage

YOUR BACKSTAGE PASS TO **LIVE!** THE MICROSOFT PLATFORM

VISUAL STUDIO LIVE! IS COMING TO A CITY NEAR YOU

PICK YOUR TOUR DATES



Get an all-access look at the Microsoft Platform and practical, unbiased Developer training at **Visual Studio Live!**. Pick your tour dates and join .NET rockstars for a week of educational sessions, workshops and networking events.

HTML5 | Windows 8 | Visual Studio/.NET | Mobile | WPF/ Silverlight

There are **FOUR** tour dates / locations to choose from – pick your favorite and prepare to solve your development challenges in 2013!



Scan the QR code to register or for more event details.

vslive.com

The C# Memory Model in Theory and Practice, Part 2

Igor Ostrovsky

This is the **second article** in a two-part series that discusses the C# memory model. As explained in the first part in the December issue of *MSDN Magazine* (msdn.microsoft.com/magazine/jj863136), the compiler and the hardware may subtly transform the memory operations of a program in ways that don't affect single-threaded behavior, but can impact multi-threaded behavior. As an example, consider this method:

```
void Init() {
    _data = 42;
    _initialized = true;
}
```

If `_data` and `_initialized` are ordinary (that is, non-volatile) fields, the compiler and the processor are allowed to reorder the operations so that `Init` executes as if it were written like this:

```
void Init() {
    _initialized = true;
    _data = 42;
}
```

In the previous article, I described the abstract C# memory model. In this article, I'll explain how the C# memory model is actually implemented on different architectures supported by the Microsoft .NET Framework 4.5.

This article discusses:

- Compiler optimizations
- C# memory model on the x86-x64 architecture
- C# memory model on the Itanium architecture
- C# memory model on the ARM architecture

Technologies discussed:

C#, x86-x64, Itanium, ARM, Microsoft .NET Framework 4.5

Compiler Optimizations

As mentioned in the first article, the compiler might optimize the code in a way that reorders memory operations. In the .NET Framework 4.5, the `csc.exe` compiler that compiles C# to IL doesn't do many optimizations, so it won't reorder memory operations. However, the just-in-time (JIT) compiler that converts IL to machine code will, in fact, perform some optimizations that reorder memory operations, as I'll discuss.

Loop Read Hoisting Consider the polling loop pattern:

```
class Test
{
    private bool _flag = true;

    public void Run()
    {
        // Set _flag to false on another thread
        new Thread(() => { _flag = false; }).Start();

        // Poll the _flag field until it is set to false
        while (_flag) ;

        // The loop might never terminate!
    }
}
```

In this case, the .NET 4.5 JIT compiler might rewrite the loop like this:

```
if (_flag) { while (true); }
```

In the single-threaded case, this transformation is entirely legal and, in general, hoisting a read out of a loop is an excellent optimization. However, if the `_flag` is set to false on another thread, the optimization can cause a hang.

Note that if the `_flag` field were volatile, the JIT compiler would not hoist the read out of the loop. (See the "Polling Loop" section in the December article for a more detailed explanation of this pattern.)

Read Elimination Another compiler optimization that can cause errors in multi-threaded code is illustrated in this sample:

```
class Test
{
    private int _A, _B;

    public void Foo()
    {
        int a = _A;
        int b = _B;
        ...
    }
}
```

The class contains two non-volatile fields, `_A` and `_B`. Method `Foo` first reads field `_A` and then field `_B`. However, because the fields are non-volatile, the compiler is free to reorder the two reads. So, if the correctness of the algorithm depends on the order of the reads, the program contains a bug.

It's hard to imagine what the compiler would gain by switching the order of the reads. Given the way `Foo` is written, the compiler probably wouldn't swap the order of the reads.

However, the reordering does happen if I add another innocuous statement at the top of the `Foo` method:

```
public bool Foo()
{
    if (_B == -1) throw new Exception(); // Extra read

    int a = _A;
    int b = _B;
    return a > b;
}
```

On the first line of the `Foo` method, the compiler loads the value of `_B` into a register. Then, the second load of `_B` simply uses the value that's already in the register instead of issuing a real load instruction.

Effectively, the compiler rewrites the `Foo` method as follows:

```
public bool Foo()
{
    int b = _B;
    if (b == -1) throw new Exception(); // Extra read

    int a = _A;
    return a > b;
}
```

Although this code sample gives a rough approximation of how the compiler optimizes the code, it's also instructive to look at the disassembly of the code:

```
if (_B == -1) throw new Exception();
push     eax
mov     edx, dword ptr [ecx+8]
// Load field _B into EDX register
cmp     edx, 0FFFFFFFh
je      00000016

int a = _A;
mov     eax, dword ptr [ecx+4]
// Load field _A into EAX register

return a > b;
cmp     eax, edx
// Compare registers EAX and EDX
...
```

Even if you don't know assembly, what's happening here is pretty easy to understand. As a part of evaluating the condition `_B == -1`, the compiler loads the `_B` field into the `EDX` register. Later, when field `_B` is read again, the compiler simply reuses the value it already has in `EDX` instead of issuing a real memory read. Consequently, the reads of `_A` and `_B` get reordered.

In this case, the correct solution is to mark field `_A` as volatile. If that's done, the compiler shouldn't reorder the reads of `_A` and

`_B`, because the load of `_A` has load-acquire semantics. However, I should point out that the .NET Framework, through version 4, doesn't handle this case correctly and, in fact, marking the `_A` field as volatile will not prevent the read reordering. This issue has been fixed in the .NET Framework version 4.5.

Read Introduction As I just explained, the compiler sometimes fuses multiple reads into one. The compiler can also split a single read into multiple reads. In the .NET Framework 4.5, read introduction is much less common than read elimination and occurs only in very rare, specific circumstances. However, it does sometimes happen.

To understand read introduction, consider the following example:

```
public class ReadIntro {
    private Object _obj = new Object();

    void PrintObj() {
        Object obj = _obj;
        if (obj != null) {
            Console.WriteLine(obj.ToString());
            // May throw a NullReferenceException
        }
    }

    void Uninitialize() {
        _obj = null;
    }
}
```

If you examine the `PrintObj` method, it looks like the `obj` value will never be null in the `obj.ToString` expression. However, that line of code could in fact throw a `NullReferenceException`. The CLR JIT might compile the `PrintObj` method as if it were written like this:

```
void PrintObj() {
    if (_obj != null) {
        Console.WriteLine(_obj.ToString());
    }
}
```

Because the read of the `_obj` field has been split into two reads of the field, the `ToString` method may now be called on a null target.

Note that you won't be able to reproduce the `NullReferenceException` using this code sample in the .NET Framework 4.5 on x86-x64. Read introduction is very difficult to reproduce in the .NET Framework 4.5, but it does nevertheless occur in certain special circumstances.

C# Memory Model Implementation on the x86-x64

As the x86 and x64 have the same behavior with respect to the memory model, I'll consider both processor variants together.

Unlike some architectures, the x86-x64 processor provides fairly strong ordering guarantees on memory operations. In fact, the JIT compiler doesn't need to use any special instructions on the x86-x64 to achieve volatile semantics; ordinary memory operations already provide those semantics. Even so, there are still specific cases when the x86-x64 processor does reorder memory operations.

x86-x64 Memory Reordering Even though the x86-x64 processor provides fairly strong ordering guarantees, a particular kind of hardware reordering still happens.

The x86-x64 processor will not reorder two writes, nor will it reorder two reads. However, the one (and only) possible reordering effect is that when a processor writes a value, that value will not be made immediately available to other processors. **Figure 1** shows an example that demonstrates this behavior.

Consider the case when methods `ThreadA` and `ThreadB` are called from different threads on a new instance of `StoreBufferExample`, as

shown in **Figure 2**. If you think about the possible outcomes of the program in **Figure 2**, three cases seem to be possible:

1. Thread 1 completes before Thread 2 starts. The outcome is `A_Won=true, B_Won=false`.
2. Thread 2 completes before Thread 1 starts. The outcome is `A_Won=false, B_Won=true`.
3. The threads interleave. The outcome is `A_Won=false, B_Won=false`.

But, surprisingly, there's a fourth case: It's possible that both the `A_Won` and `B_Won` fields will be true after this code has finished! Because of the store buffer, stores can get "delayed," and therefore end up reordered with a subsequent load. Even though this outcome is not consistent with any interleaving of Thread 1 and Thread 2 executions, it can still happen.

This example is interesting because we have a processor (the x86-x64) with relatively strong ordering, and all fields are volatile—and we *still* observe a reordering of memory operations. Even though the write to `A` is volatile and the read from `A_Won` is also volatile, the fences are both one-directional, and in fact allow this reordering. So, the `ThreadA` method may effectively execute as if it were written like this:

```
public void ThreadA()
{
    bool tmp = B;
    A = true;
    if (!tmp) A_Won = 1;
}
```

One possible fix is to insert a memory barrier into *both* `ThreadA` and `ThreadB`. The updated `ThreadA` method would look like this:

```
public void ThreadA()
{
    A = true;
    Thread.MemoryBarrier();
    if (!B) aWon = 1;
}
```

The CLR JIT will insert a "lock or" instruction in place of the memory barrier. A locked x86 instruction has the side effect of flushing the store buffer:

```
mov     byte ptr [ecx+4],1
lock or  dword ptr [esp],0
cmp     byte ptr [ecx+5],0
jne     00000013
mov     byte ptr [ecx+6],1
ret
```

As an interesting side note, the Java programming language takes a different approach. The Java memory model has a slightly stronger definition of "volatile" that doesn't permit store-load reordering, so a Java compiler on the x86 will typically emit a locked instruction after a volatile write.

x86-x64 Remarks The x86 processor has a fairly strong memory model, and the only source of reordering at the hardware level is the store buffer. The store buffer can cause a write to get reordered with a subsequent read (store-load reordering).

Also, certain compiler optimizations can result in reordering of memory operations. Notably, if several reads access the same memory location, the compiler might choose to perform the read only once and keep the value in a register for subsequent reads.

One interesting piece of trivia is that the C# volatile semantics closely match the hardware reordering guarantees made by x86-x64 hardware. As a result, reads and writes of volatile fields require no special instructions on the x86: Ordinary reads and writes (for

Figure 1 StoreBufferExample

```
class StoreBufferExample
{
    // On x86 .NET Framework 4.5, it makes no difference
    // whether these fields are volatile or not
    volatile int A = 0;
    volatile int B = 0;
    volatile bool A_Won = false;
    volatile bool B_Won = false;

    public void ThreadA()
    {
        A = true;
        if (!B) A_Won = true;
    }

    public void ThreadB()
    {
        B = true;
        if (!A) B_Won = true;
    }
}
```

example, using the `MOV` instruction) are sufficient. Of course, your code shouldn't depend on these implementation details because they vary among hardware architectures and possibly .NET versions.

C# Memory Model Implementation on the Itanium Architecture

The Itanium hardware architecture has a memory model weaker than that of the x86-x64. Itanium was supported by the .NET Framework until version 4.

Even though Itanium is no longer supported in the .NET Framework 4.5, understanding the Itanium memory model is useful when you read older articles on the .NET memory model and have to maintain code that incorporated recommendations from those articles.

Itanium Reordering Itanium has a different instruction set than the x86-x64, and memory model concepts show up in the instruction set. Itanium distinguishes between an ordinary load (`LD`) and load-acquire (`LD.ACQ`), and an ordinary store (`ST`) and store-release (`ST.REL`).

Ordinary loads and stores can be freely reordered by the hardware, as long as the single-threaded behavior doesn't change. For example, look at this code:

```
class ReorderingExample
{
    int _a = 0, _b = 0;

    void PrintAB()
    {
        int a = _a;
        int b = _b;
        Console.WriteLine("A:{0} B:{1}", a, b);
    }
    ...
}
```

Consider two reads of `_a` and `_b` in the `PrintAB` method. Because the reads access an ordinary, non-volatile field, the compiler will use ordinary `LD` (not `LD.ACQ`) to implement the reads. Consequently, the two reads might be effectively reordered in the hardware, so that `PrintAB` behaves as if it were written like this:

```
void PrintAB()
{
    int b = _b;
    int a = _a;
    Console.WriteLine("A:{0} B:{1}", a, b);
}
```

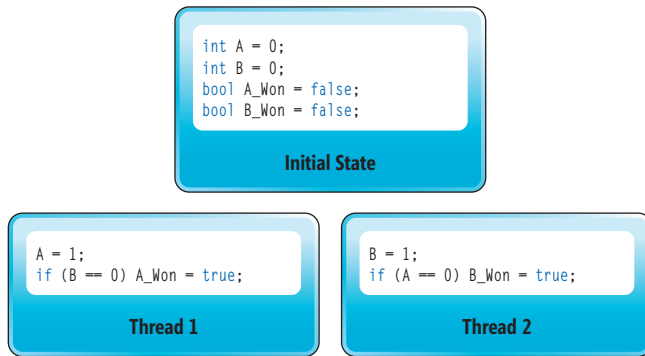


Figure 2 Calling ThreadA and ThreadB Methods from Different Threads

In practice, whether the reordering happens or not depends on a variety of unpredictable factors—what’s in the processor cache, how busy the processor pipeline is and so forth. However, the processor will not reorder two reads if they’re related via data dependency. Data dependency between two reads occurs when the value returned by a memory read determines the location of the read by a subsequent read.

This example illustrates data dependency:

```

class Counter { public int _value; }

class Test
{
    private Counter _counter = new Counter();

    void Do()
    {
        Counter c = _counter; // Read 1
        int value = c._value; // Read 2
    }
}

```

In the Do method, Itanium will never reorder Read 1 and Read 2, even though Read 1 is an ordinary load and not load-acquire. It might seem obvious that these two reads can’t be reordered: The first read determines which memory location the second read

Figure 3 Lazy Initialization

```

// Warning: Might not work on future architectures and .NET versions;
// do not use
class LazyExample
{
    private BoxedInt _boxedInt;

    int GetInt()
    {
        BoxedInt b = _boxedInt; // Read 1
        if (b == null)
        {
            lock(this)
            {
                if (_boxedInt == null)
                {
                    b = new BoxedInt();
                    b._value = 42; // Write 1
                    _boxedInt = b; // Write 2
                }
            }
        }

        int value = b._value; // Read 2
        return value;
    }
}

```

should access! However, some processors—other than Itanium—may in fact reorder the reads. The processor might guess on the value that Read 1 will return and perform Read 2 *speculatively*, even before Read 1 has completed. But, again, Itanium will not do that.

I’ll get back to the data dependency in Itanium discussion in a bit, and its relevance to the C# memory model will become clearer.

Also, itanium will not reorder two ordinary reads if they’re related via control dependency. Control dependency occurs when the value returned by a read determines whether a subsequent instruction will execute.

So, in this example, the reads of `_initialized` and `_data` are related via control dependency:

```

void Print() {
    if (_initialized) // Read 1
        Console.WriteLine(_data); // Read 2
    else
        Console.WriteLine("Not initialized");
}

```

Even if `_initialized` and `_data` are ordinary (non-volatile) reads, the Itanium processor will not reorder them. Note that the JIT compiler is still free to reorder the two reads, and in some cases will.

Also, it’s worth pointing out that, like the x86-x64 processor, Itanium also uses a store buffer, so the StoreBufferExample shown in **Figure 1** will exhibit the same kind of reorderings on Itanium as it did on the x86-x64. An interesting piece of trivia is that if you use LD.ACQ for all reads and ST.REL for all writes on Itanium, you basically get the x86-x64 memory model, where the store buffer is the only source of reordering.

Compiler Behavior on Itanium The CLR JIT compiler has one surprising behavior on Itanium: all writes are emitted as ST.REL, and not ST. Consequently, a volatile write and a non-volatile write will typically emit the same instruction on Itanium. However, an ordinary read will be emitted as LD; only reads from volatile fields are emitted as LD.ACQ.

This behavior might come as a surprise because the compiler is certainly not required to emit ST.REL for non-volatile writes. As far as the European Computer Manufacturers Association (ECMA) C# specification is concerned, the compiler could emit ordinary ST instructions. Emitting ST.REL is just something extra that the compiler chooses to do, in order to ensure that a particular common (but in theory incorrect) pattern will work as expected.

It can be difficult to imagine what that important pattern might be where ST.REL must be used for writes, but LD is sufficient for reads. In the PrintAB example presented earlier in this section, constraining just the writes wouldn’t help, because reads could still be reordered.

There’s one very important scenario in which using ST.REL with ordinary LD is sufficient: when the loads themselves are ordered using data dependency. This pattern comes up in lazy initialization, which is an extremely important pattern. **Figure 3** shows an example of lazy initialization.

In order for this bit of code to always return 42—even if GetInt is called from multiple threads concurrently—Read 1 must not be reordered with Read 2, and Write 1 must not be reordered with Write 2. The reads will not be reordered by the Itanium processor because they’re related via data dependency. And, the writes will not be reordered because the CLR JIT emits them as ST.REL.

Figure 4 A Correct Implementation of Lazy Initialization

```
class BoxedInt
{
    public int _value;
    public BoxedInt() { }
    public BoxedInt(int value) { _value = value; }
}

class LazyExample
{
    private volatile BoxedInt _boxedInt;

    int GetInt()
    {
        BoxedInt b = _boxedInt;
        if (b == null)
        {
            b = new BoxedInt(42);
            _boxedInt = b;
        }

        return b._value;
    }
}
```

Note that if the `_boxedInt` field were volatile, the code would be correct according to the ECMA C# spec. That's the best kind of correct, and arguably the only real kind of correct. However, even if `_boxed` is not volatile, the current version of the compiler will ensure that the code still works on Itanium in practice.

Of course, loop read hoisting, read elimination and read introduction may be performed by the CLR JIT on Itanium, just as they are on x86-x64.

Itanium Remarks The reason Itanium is an interesting part of the story is that it was the first architecture with a weak memory model that ran the .NET Framework.

As a result, in a number of articles about the C# memory model and the volatile keyword and C#, the authors generally had Itanium in mind. After all, until the .NET Framework 4.5, Itanium was the only architecture other than the x86-x64 that ran the .NET Framework.

Consequently, the author might say something like, "In the .NET 2.0 memory model, all writes are volatile—even those to non-volatile fields." What the author means is that on Itanium, CLR will emit all writes as `ST.REL`. This behavior isn't guaranteed by the ECMA C# spec, and, consequently, might not hold in future versions of the .NET Framework and on future architectures (and, in fact, does not hold in the .NET Framework 4.5 on ARM).

Figure 5 An Incorrect Implementation of Lazy Initialization

```
// Warning: Bad code
class LazyExample
{
    private BoxedInt _boxedInt; // Note: This field is not volatile

    int GetInt()
    {
        BoxedInt b = _boxedInt; // Read 1
        if (b == null)
        {
            b = new BoxedInt(42); // Write 1 (inside constructor)
            _boxedInt = b;        // Write 2
        }

        return b._value;        // Read 2
    }
}
```

Similarly, some folks would argue that lazy initialization is correct in the .NET Framework even if the holding field is non-volatile, while others would say that the field must be volatile.

And of course, developers wrote code against these (sometimes contradictory) assumptions. So, understanding the Itanium part of the story can be helpful when trying to make sense of concurrent code written by someone else, reading older articles or even just talking to other developers.

C# Memory Model Implementation on ARM

The ARM architecture is the most recent addition to the list of architectures supported by the .NET Framework. Like Itanium, ARM has a weaker memory model than the x86-x64.

ARM Reordering Just like Itanium, ARM is allowed to freely reorder ordinary reads and writes. However, the solution that ARM provides to tame the movement of reads and writes is somewhat different from that of Itanium. ARM exposes a single instruction—DMB—that acts as a full memory barrier. No memory operation can pass over DMB in either direction.

In addition to the constraints imposed by the DMB instruction, ARM also respects data dependency, but doesn't respect control dependency. See the "Itanium Reordering" section earlier in this article for explanations of data and control dependencies.

Compiler Behavior on ARM The DMB instruction is used to implement the volatile semantics in C#. On ARM, the CLR JIT implements a read from a volatile field using an ordinary read (such as `LDR`) followed by the DMB instruction. Because the DMB instruction will prevent the volatile read from getting reordered with any subsequent operations, this solution correctly implements the acquire semantics.

A write to a volatile field is implemented using the DMB instruction followed by an ordinary write (such as `STR`). Because the DMB instruction prevents the volatile write from getting reordered with any prior operations, this solution correctly implements the release semantics.

Just as with the Itanium processor, it would be nice to go beyond the ECMA C# spec and keep the lazy initialization pattern working, because a lot of existing code depends on it. However, making all writes effectively volatile is not a good solution on ARM because the DBM instruction is fairly costly.

In the .NET Framework 4.5, the CLR JIT uses a slightly different trick to get lazy initialization working. The following are treated as "release" barriers:

1. Writes to reference-type fields on the garbage collector (GC) heap
2. Writes to reference-type static fields

As a result, any write that might publish an object is treated as a release barrier.

This is the relevant part of `LazyExample` (recall that none of the fields are volatile):

```
b = new BoxedInt();
b._value = 42; // Write 1
// DMB will be emitted here
_bboxedInt = b; // Write 2
```

Because the CLR JIT emits the DMB instructions prior to the publication of the object into the `_boxedInt` field, Write 1 and Write 2 will not be reordered. And because ARM respects data

dependence, the reads in the lazy initialization pattern will not be reordered either, and the code will work correctly on ARM.

So, the CLR JIT makes an extra effort (beyond what's mandated in the ECMA C# spec) to keep the most common variant of incorrect lazy initialization working on ARM.

As a final comment on ARM, note that—as on x86-x64 and Itanium—loop read hoisting, read elimination and read introduction are all legitimate optimizations as far as the CLR JIT is concerned.

Example: Lazy Initialization

It can be instructive to look at a few different variants of the lazy initialization pattern and think about how they'll behave on different architectures.

Correct Implementation The implementation of lazy initialization in **Figure 4** is correct according to the C# memory model as defined by the ECMA C# spec, and so it's guaranteed to work on all architectures supported by current and future versions of the .NET Framework.

Note that even though the code sample is correct, in practice it's still preferable to use the `Lazy<T>` or the `LazyInitializer` type.

Incorrect Implementation No. 1 **Figure 5** shows an implementation that *isn't* correct according to the C# memory model. In spite of this, the implementation will probably work on the x86-x64, Itanium and ARM in the .NET Framework. This version of the code is not correct. Because `_boxedInt` isn't volatile, a C#

compiler is permitted to reorder Read 1 with Read 2, or Write 1 with Write 2. Either reordering would potentially result in 0 returned from `GetInt`.

However, this code will behave correctly (that is, always return 42) on all architectures in the .NET Framework versions 4 and 4.5:

- x86-x64:
 - Writes and reads will not be reordered. There is no store-load pattern in the code, and also no reason the compiler would cache values in registers.
- Itanium:
 - Writes will not be reordered because they are `ST.REL`.
 - Reads will not be reordered due to data dependency.
- ARM:
 - Writes will not be reordered because DMB is emitted before “`_boxedInt = b`”.
 - Reads will not be reordered due to data dependency.

Of course, you should use this information only to try to understand the behavior of existing code. *Do not* use this pattern when writing new code.

Incorrect Implementation No. 2 The incorrect implementation in **Figure 6** may fail on both ARM and Itanium.

This version of lazy initialization uses two separate fields to track the data (`_value`) and whether the field is initialized (`_initialized`). As a result, the two reads—Read 1 and Read 2—are no longer related via data dependency. Additionally, on ARM, the writes might also get reordered, for the same reasons as in the next incorrect implementation (No. 3).

As a result, this version may fail and return 0 on ARM and Itanium in practice. Of course, `GetInt` is allowed to return 0 on x86-x64 (and also as a result of JIT optimizations), but that behavior doesn't seem to happen in the .NET Framework 4.5.

Incorrect Implementation No. 3 Finally, it's possible to get the example to fail even on x86-x64. I just have to add one innocuous-looking read, as shown in **Figure 7**.

The extra read that checks whether `_value < 0` can now cause the compiler to cache the value in register. As a result, Read 2 will get serviced from a register, and so it gets effectively reordered with Read 1. Consequently, this version of `GetInt` may in practice return 0 even on x86-x64.

Wrapping Up

When writing new multi-threaded code, it's generally a good idea to avoid the complexity of the C# memory model altogether by using high-level concurrency primitives like locks, concurrent collections, tasks, and parallel loops. When writing CPU-intensive code, it sometimes makes sense to use volatile fields, as long as you only rely on the ECMA C# specification guarantees and not on architecture-specific implementation details. ■

IGOR OSTROVSKY is a senior software development engineer at Microsoft. He has worked on Parallel LINQ, the Task Parallel Library, and other parallel libraries and primitives in the .NET Framework. Ostrovsky blogs about programming topics at igoro.com.

THANKS to the following technical experts for reviewing this article:

Joe Duffy, Eric Eilebrecht, Joe Hoag, Emad Omara, Grant Richins, Jaroslav Sevcik and Stephen Toub

Figure 6 A Second Incorrect Implementation of Lazy Initialization

```
// Warning: Bad code
class LazyExample
{
    private int _value;
    private bool _initialized;

    int GetInt()
    {
        if (!_initialized) // Read 1
        {
            _value = 42;
            _initialized = true;
        }

        return _value;    // Read 2
    }
}
```

Figure 7 A Third Incorrect Implementation of Lazy Initialization

```
// Warning: Bad code
class LazyExample
{
    private int _value;
    private bool _initialized;

    int GetInt()
    {
        if (_value < 0) throw new Exception(); // Note: extra reads to get _value
                                                // pre-loaded into a register

        if (!_initialized) // Read 1
        {
            _value = 42;
            _initialized = true;
            return _value;
        }

        return _value;    // Read 2
    }
}
```

Building Hypermedia Web APIs with ASP.NET Web API

Pablo Cibraro

Hypermedia—better known as Hypermedia as the Engine of Application State (HATEOAS)—is one of the main constraints of Representational State Transfer (REST). The idea is that hypermedia artifacts, such as links or forms, can be used to describe how clients can interact with a set of HTTP services. This has quickly become an interesting concept for developing evolvable API design. This is not any different from how we usually interact with the Web. We typically remember a single entry point or URL for the homepage of a Web site, and later move through the different sections of the site using links. We also use forms, which come with a predefined action or URL to submit data that the site might need to perform some action.

This article discusses:

- How hypermedia can help make Web APIs evolvable
- Making hypermedia artifacts available using media types
- Hypertext Application Language
- Supporting hypermedia in ASP.NET Web API

Technologies discussed:

ASP.NET, ASP.NET Web API

Code download available at:

archive.msdn.microsoft.com/mag201301Hypermedia

Developers have a tendency to provide static descriptions of all the supported methods in a service, ranging from formal contracts such as Web Services Description Language (WSDL) in SOAP services to simple documentation in non-hypermedia Web APIs. The main problem with this is that a static API description couples clients heavily to the server. In a nutshell, it inhibits evolvability, as any change in the API description can break all existing clients.

This was no problem for a while in the enterprise, where the number of client applications could be controlled and known in advance. However, when the number of potential clients increases exponentially—as is the case today, with thousands of third-party applications running across multiple devices—it's a bad idea. Still, simply moving from SOAP to HTTP services is no guarantee that the problem will be solved. If there is some knowledge on the client for computing URLs, for example, the problem still exists, even without any explicit contract such as WSDL. Hypermedia is what provides the ability to shield clients from any server changes.

The application state workflow, which determines what the client can do next, should also be located on the server side. Suppose an action in a resource is available only for a given state; should that logic reside in any possible API client? Definitely not. The server should always mandate what can be done with a resource. For example, if a purchase order (PO) is canceled, the client application shouldn't be allowed to submit that PO, which means a link or a form to submit the PO should not be available in the response sent to the client.

Figure 1 Using XHTML to Expose a List of Products

```
<div id="products">
  <ul class="all">
    <li>
      <span class="product-id">1</span>
      <span class="product-name">Product 1</span>
      <span class="product-price">$5.34</span>
      <a rel="add-cart" href="/cart" type="application/xml"/>
    </li>
    <li>
      <span class="product-id">2</span>
      <span class="product-name">Product 2</span>
      <span class="product-price">$10</span>
      <a rel="add-cart" href="/cart" type="application/xml"/>
    </li>
  </ul>
</div>
```

Hypermedia to the Rescue

Linking has always been a key component of REST architecture. Of course, links are familiar in UI contexts such as browsers; for example, consider a “See details” link for obtaining the details of a given product in a catalog. But what about computer-to-computer scenarios where there’s no UI or user interaction? The idea is that you can use hypermedia artifacts in these scenarios as well.

Linking has always been a key component of REST architecture.

With this new approach, the server doesn’t only return data. It returns data and hypermedia artifacts. The hypermedia artifacts give the client a way to determine the available set of actions that can be performed at a given point based on the state of the server application workflow.

This is one area that typically differentiates a regular Web API from a RESTful API, but there are other constraints that also apply, so discussing whether an API is RESTful or not probably doesn’t make sense in most cases. What matters is that the API uses HTTP correctly as an application protocol and leverages hypermedia when possible. By enabling hypermedia, you can create self-discoverable APIs. This is no excuse for not providing documentation, but the APIs are more flexible in terms of updatability.

Which hypermedia artifacts are available is mainly determined by the chosen media types. Many of the media types we use nowadays for building a Web API, such as JSON or XML, don’t have a built-in concept for representing links or forms, as HTML does.

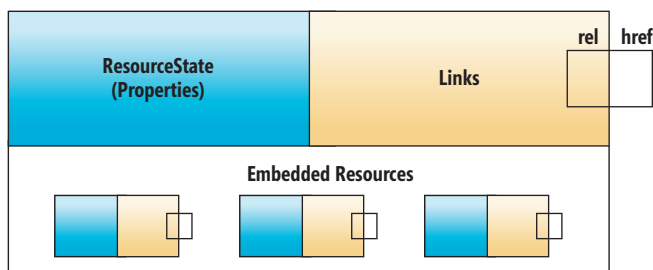


Figure 2 The HAL Media Type

You can leverage those media types by defining a way to express hypermedia, but this requires that clients understand how the hypermedia semantics are defined on top of those. In contrast, media types such as XHTML (application/xhtml+xml) or ATOM (application/atom+xml) already support some of these hypermedia artifacts such as links or forms.

In the case of HTML, a link is made up of three components: an “href” attribute pointing to a URL, a “rel” attribute for describing how the link relates to the current resource, and an optional “type” attribute for specifying the expected media type. For example, if you want to expose a list of products in a catalog using XHTML, the resource payload might look like what’s shown in **Figure 1**.

In this example, the catalog of products is represented with standard HTML elements, but I used XHTML because it’s much easier to parse with any existing XML library. Also, as part of the payload, an anchor (a) element has been included, representing a link for adding the item to the current user cart. By looking at the link, a client can infer its usage via the rel attribute (add a new item), and use the href for performing an operation over that resource (/cart). It’s important to notice that the links are generated by the server based on its business workflow, so the client doesn’t need to hardcode any URL or infer any rule. This also offers new opportunities to modify the workflow during runtime without affecting the existing clients at all. If any of the products offered in the catalog are out of stock, the server can simply omit the link for adding that product to the cart. From the client perspective, the link isn’t available so the product can’t be ordered. More complex rules related to that workflow might apply on the server side, but the client is totally unaware of that, as the only thing that matters is that the link isn’t present. Thanks to hypermedia and links, the client has been decoupled from the business workflow on the server side.

Many of the media types we use nowadays for building a Web API, such as JSON or XML, don’t have a built-in concept for representing links or forms, as HTML does.

Moreover, the evolvability of an API design can be improved with hypermedia and links. As the business workflow on the server evolves, it can offer additional links for new functionality. In our product catalog example, the server might include a new link for marking a product as a favorite, like this:

```
<li>
  <span class="product-id">1</span>
  <span class="product-name">Product 1</span>
  <span class="product-price">$5.34</span>
  <a rel="add-cart" href="/cart/1" type="application/xml"/>
  <a rel="favorite" href="/product_favorite/1" type="application/xml"/>
</li>
```


Figure 3 The Product Catalog in HAL

```
<resource href="/products">
  <link rel="next" href="/products?page=2" />
  <link rel="find" href="/products?id" templated="true" />
  <resource rel="product" href="/products/1">
    <link rel="add-cart" href="/cart/" />
    <name>Product 1</name>
    <price>5.34</price>
  </resource>
  <resource rel="product" href="/products/2">
    <link rel="add-cart" href="/cart/" />
    <name>Product 2</name>
    <price>10</price>
  </resource>
</resource>
```

While existing clients might ignore that link and remain unaffected by this new functionality, newer clients can start using it right away. In this way, it wouldn't be crazy to think of having a single entry point or root URL for your Web API that contains links to discover the rest of the functionality. For example, you could have a single URL `"/shopping_cart"` that returns the following HTML representation:

```
<div class="root">
  <a rel="products" href="/products"/>
  <a rel="cart" href="/cart"/>
  <a rel="favorites" href="/product_favorite"/>
</div>
```

Analogous functionality is also found in OData services, which expose a single service document in the root URL with all the supported resource sets and links for getting the data associated to them.

Links represent a great way to connect servers and clients, but there's an evident problem with them. In the previous example with the product catalog, a link in HTML offers only the `rel`, `href` and type attributes, which implies some out-of-band knowledge about what to do with that URL expressed in the `href` attribute. Should the client use an HTTP POST or HTTP GET? If it uses a POST, what data should the client include in the request body? While all that knowledge could be documented somewhere, wouldn't it be great if the clients could actually discover that functionality? For all these questions, using HTML forms is the answer that makes a lot of sense.

Figure 4 The JSON Representation for the Sample Resource

```
{
  "_links": {
    "self": { "href": "/products" },
    "next": { "href": "/products?page=2" },
    "find": { "href": "/products?id", "templated": true }
  },
  "_embedded": {
    "products": [
      {
        "_links": {
          "self": { "href": "/products/1" },
          "add-cart": { "href": "/cart/" }
        },
        "name": "Product 1",
        "price": 5.34
      },
      {
        "_links": {
          "self": { "href": "/products/2" },
          "add-cart": { "href": "/cart/" }
        },
        "name": "Product 2",
        "price": 10
      }
    ]
  }
}
```

Forms in Action

When you interact with the Web using a browser, actions are typically represented with forms. In the product catalog example, pressing the Add to Cart link implies an HTTP GET sent to the server to return an HTML form that can be used to add the product to the cart. That form could contain an `“action”` attribute with a URL, a `“method”` attribute representing the HTTP method, and some input fields that might require input from the user, as well as some readable instructions to move forward.

When you interact with the Web
using a browser, actions are
typically represented with forms.

You can do the same for a machine-to-machine scenario. Rather than having a human being interacting with a form, you might have an application running JavaScript or C#. In the product catalog, an HTTP GET to the `“add-cart”` link for the first product would retrieve the following form represented in XHTML:

```
<form action="/cart" method="POST">
  <input type="hidden" id="product-id">1</input>
  <input type="hidden" id="product-price">5.34</input>
  <input type="hidden" id="product-quantity" class="required">1</input>
  <input type="hidden" id="__forgeryToken">XXXXXXX</input>
</form>
```

The client application has now been decoupled from certain details related to adding the product to the cart. It just needs to submit this form using an HTTP POST to the URL specified in the action attribute. The server can also include additional information in the form—for example, a forgery token to avoid cross-site request forgery (CSRF) attacks or to sign the data that's prepopulated for the server.

This model allows any Web API to freely evolve by offering new forms based on different factors such as user permissions or the version the client wants to use.

Hypermedia for XML and JSON?

As I mentioned earlier, the generic media types for XML (`application/xml`) and JSON (`application/json`) don't have a built-in support for hypermedia links or forms. While it's possible to extend those

Figure 5 The MediaTypeFormatter Class

```
public abstract class MediaTypeFormatter
{
    public Collection<Encoding> SupportedEncodings { get; }

    public Collection<MediaTypeHeaderValue> SupportedMediaTypes { get; }

    public abstract bool CanReadType(Type type);

    public abstract bool CanWriteType(Type type);

    public virtual Task<object> ReadFromStreamAsync(Type type, Stream readStream,
        HttpContent content, IFormatterLogger formatterLogger);

    public virtual Task WriteToStreamAsync(Type type, object value,
        Stream writeStream, HttpContent content, TransportContext transportContext);
}
```

media types with domain-specific concepts such as “application/vnd-shoppingcart+xml,” this requires new clients to understand all the semantics defined in that new type (and it would probably also generate a proliferation of media types) so it’s generally not considered a good idea.

For that reason, a new media type that extends XML and JSON with linking semantics and is called Hypertext Application Language (HAL) has been proposed. The draft, which simply defines a standard way to express hyperlinks and embedded resources (data) using XML and JSON, is available at stateless.co/hal_specification.html. The HAL media type defines a resource that contains a set of properties, a set of links and a set of embedded resources, as shown in **Figure 2**.

The generic media types for XML (application/xml) and JSON (application/json) don’t have a built-in support for hypermedia links or forms.

Figure 3 shows an example of how a product catalog would look in HAL using both the XML and JSON representations. **Figure 4** is the JSON representation for the sample resource.

Supporting Hypermedia in ASP.NET Web API

So far I’ve discussed some of the theory behind hypermedia in the design of Web APIs. Now let’s see how that theory can actually be implemented in the real world using ASP.NET Web API, with all the extensibility points and features this framework provides.

At a core level, ASP.NET Web API supports the idea of formatters. A formatter implementation knows how to deal with a specific media

type, and how to serialize or deserialize it into concrete .NET types. In the past, support for new media types was very limited in ASP.NET MVC. Only HTML and JSON were treated as first-class citizens and fully supported across the entire stack. Furthermore, there was no consistent model for supporting content negotiation. You could support different media type formats for the response messages by providing custom ActionResult implementations, but it wasn’t clear how a new media type could be introduced for deserializing request messages. This was typically solved by leveraging the model-binding infrastructure with new model binders or value providers. Fortunately, this inconsistency has been solved in ASP.NET Web API with the introduction of formatters.

Every formatter derives from the base class `System.Net.Http.Formatting.MediaTypeFormatter` and overrides the method `CanReadType/ReadFromStreamAsync` for supporting deserialization and the method `CanWriteType/WriteToStreamAsync` for supporting serialization of .NET types into a given media type format.

Figure 5 shows the definition of the `MediaTypeFormatter` class.

Formatters play a very important role in ASP.NET Web API in supporting content negotiation, as the framework can now choose the correct formatter based on the values received in the “Accept” and “Content-Type” headers of the request message.

The `ReadFromStreamAsync` and `WriteToStreamAsync` methods rely on the Task Parallel Library (TPL) for doing the asynchronous work, so they return a Task instance. In case you want to explicitly make your formatter implementation work synchronously, the base class, `BufferedMediaTypeFormatter`, does it for you internally. This base class provides two methods you can override in an implementation, `SaveToStream` and `ReadFromStream`, which are the synchronous versions of `SaveToStreamAsync` and `ReadFromStreamAsync`.

A formatter implementation knows how to deal with a specific media type, and how to serialize or deserialize it into concrete .NET types.

Developing a MediaTypeFormatter for HAL

HAL uses specific semantics for representing resources and links, so you can’t use just any model in a Web API implementation. For that reason, one base class for representing a resource and another for a collection of resources are used to make the implementation of the formatter much simpler:

```
public abstract class LinkedResource
{
    public List<Link> Links { get; set; }
    public string HRef { get; set; }
}

public abstract class LinkedResourceCollection<T> : LinkedResource,
    ICollection<T> where T : LinkedResource
{
    // Rest of the collection implementation
}
```

Figure 6 The BufferedMediaTypeFormatter Base Class

```
public class HalXmlMediaTypeFormatter : BufferedMediaTypeFormatter
{
    public HalXmlMediaTypeFormatter()
        : base()
    {
        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue(
            "application/hal+xml"));
    }

    public override bool CanReadType(Type type)
    {
        return type.BaseType == typeof(LinkedResource) ||
            type.BaseType.GetGenericTypeDefinition() ==
                typeof(LinkedResourceCollection<>);
    }

    public override bool CanWriteType(Type type)
    {
        return type.BaseType == typeof(LinkedResource) ||
            type.BaseType.GetGenericTypeDefinition() ==
                typeof(LinkedResourceCollection<>);
    }

    ...
}
```

Figure 7 The WriteToStream and ReadFromStream Methods

```
public override void WriteToStream(Type type, object value,
    System.IO.Stream writeStream, System.Net.Http.HttpContent content)
{
    var encoding = base.SelectCharacterEncoding(content.Headers);

    var settings = new XmlWriterSettings();
    settings.Encoding = encoding;

    var writer = XmlWriter.Create(writeStream, settings);

    var resource = (LinkResource)value;

    if (resource is IEnumerable)
    {
        writer.WriteStartElement("resource");
        writer.WriteAttributeString("href", resource.HRef);

        foreach (LinkResource innerResource in (IEnumerable)resource)
        {
            // Serializes the resource state and links recursively
            SerializeInnerResource(writer, innerResource);
        }

        writer.WriteEndElement();
    }
    else
    {
        // Serializes a single linked resource
        SerializeInnerResource(writer, resource);
    }

    writer.Flush();
    writer.Close();
}

public override object ReadFromStream(Type type,
    System.IO.Stream readStream, System.Net.Http.HttpContent content,
```

```
IFormatterLogger formatterLogger)
{
    if (type != typeof(LinkResource))
        throw new ArgumentException(
            "Only the LinkResource type is supported", "type");

    var value = (LinkResource)Activator.CreateInstance(type);

    var reader = XmlReader.Create(readStream);

    if (value is IEnumerable)
    {
        var collection = (ILinkResourceCollection)value;

        reader.ReadStartElement("resource");
        value.HRef = reader.GetAttribute("href");

        var innerType = type.BaseType.GetGenericArguments().First();

        while (reader.Read() && reader.LocalName == "resource")
        {
            // Deserializes a linked resource recursively
            var innerResource = DeserializeInnerResource(reader, innerType);
            collection.Add(innerResource);
        }
    }
    else
    {
        // Deserializes a linked resource recursively
        value = DeserializeInnerResource(reader, type);
    }

    reader.Close();

    return value;
}
```

The real model classes that the Web API controllers will use can derive from these two base classes. For example, a product or a collection of products can be implemented as follows:

Figure 8 The ProductCatalogController Class

```
public class ProductCatalogController : ApiController
{
    public static Products Products = new Products
    {
        new Product
        {
            Id = 1,
            Name = "Product 1",
            UnitPrice = 5.34M,
            Links = new List<Link>
            {
                new Link { Rel = "add-cart", HRef = "/api/cart" },
                new Link { Rel = "self", HRef = "/api/products/1" }
            }
        },
        new Product
        {
            Id = 2,
            Name = "Product 2",
            UnitPrice = 10,
            Links = new List<Link>
            {
                new Link { Rel = "add-cart", HRef = "/cart" },
                new Link { Rel = "self", HRef = "/api/products/2" }
            }
        }
    };

    public Products Get()
    {
        return Products;
    }
}
```

```
public class Product : LinkResource
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal UnitPrice { get; set; }
}

...
public class Products : LinkResourceCollection<Product>
{
}
```

Now, with a standard way to define HAL models, it's time to implement the formatter. The simplest way to start a new formatter implementation is to derive from either the `MediaTypeFormatter` base class or from the `BufferedMediaTypeFormatter` base class. The example in **Figure 6** uses the second base class.

The code first defines in the constructor the supported media types for this implementation ("application/hal+xml"), and overrides the `CanReadType` and `CanWriteType` methods to specify the supported .NET types, which have to derive from `LinkResource` or `LinkResourceCollection`. Because it was defined in the constructor, this implementation only supports the XML variant of HAL. Another formatter could optionally be implemented to support the JSON variant.

The real work is done in the `WriteToStream` and `ReadFromStream` methods, shown in **Figure 7**, which will use an `XmlWriter` and `XmlReader`, respectively, to write and read an object into and out of a stream.

The last step is to configure the formatter implementation as part of the Web API host. This step can be accomplished in almost the same way as in ASP.NET or ASP.NET Web API Self-Host, with a single difference in the needed `HttpConfiguration` implementation.

While Self-Host uses an `HttpSelfHostConfiguration` instance, ASP.NET typically uses the `HttpConfiguration` instance available globally in `System.Web.Http.GlobalConfiguration.Configuration`. The `HttpConfiguration` class provides a `Formatters` collection into which you can inject your own formatter implementation. Here's how to do this for ASP.NET:

```
protected void Application_Start()
{
    Register(GlobalConfiguration.Configuration);
}

public static void Register(HttpConfiguration config)
{
    config.Formatters.Add(new HalXmlMediaTypeFormatter());
}
```

Once the formatter is configured in the ASP.NET Web API pipeline, any controller can simply return a model class derived from `LinkResource` to be serialized by the formatter using HAL. For the product catalog example, the product and the collection of the products representing the catalog can be derived from `LinkResource` and `LinkResourceCollection`, respectively:

```
public class Product : LinkResource
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal UnitPrice { get; set; }
}

public class Products : LinkResourceCollection<Product>
{
}
```

The controller `ProductCatalogController`, which handles all the requests for the product catalog resource, can now return instances of `Product` and `Products` as shown in **Figure 8** for the `Get` method.

This example uses the HAL format, but you can also use a similar approach to build a formatter that uses Razor and templates for serializing models into XHTML. You'll find a concrete implementation of a `MediaTypeFormatter` for Razor in `RestBugs`, a sample application created by Howard Dierking to demonstrate how ASP.NET Web API can be used to create hypermedia Web APIs, at github.com/howarddierking/RestBugs.

Formatters make it easy to extend your Web API with new media types.

Better Linking Support in the Web API Controllers

Something is definitely wrong with the previous `ProductCatalogController` example. All the links have been hardcoded, which might cause a lot of headaches if the routes change often. The good news is that the framework provides a helper class called `System.Web.Http.Routing.UrlHelper` for automatically inferring the links from the routing table. An instance of this class is available in the `ApiController` base class through the `Url` property, so it can easily be used in any controller method. This is what the `UrlHelper` class definition looks like:

```
public class UrlHelper
{
    public string Link(string routeName,
        IDictionary<string, object> routeValues);
    public string Link(string routeName, object routeValues);
    public string Route(string routeName,
        IDictionary<string, object> routeValues);
    public string Route(string routeName, object routeValues);
}
```

The `Route` methods return the relative URL for a given route (for example, `/products/1`), and the `Link` methods return the absolute

Figure 9 How the `UrlHelper` Class Could Be Used in the `Get` Method

```
public Products Get()
{
    var products = GetProducts();
    foreach (var product in products)
    {
        var selfLink = new Link
        {
            Rel = "self",
            HRef = Url.Route("API Default",
                new
                {
                    controller = "ProductCatalog",
                    id = product.Id
                })
        };
        product.Links.Add(selfLink);
        if(product.IsAvailable)
        {
            var addCart = new Link
            {
                Rel = "add-cart",
                HRef = Url.Route("API Default",
                    new
                    {
                        controller = "Cart"
                    })
            };
            product.Links.Add(addCart);
        }
    }
    return products;
}
```

URL, which is the one that can be used in the models to avoid any hardcoding. The `Link` method receives two arguments: the route name and the values to compose the URL.

Figure 9 shows how, in the previous product catalog example, the `UrlHelper` class could be used in the `Get` method.

The link "self" for the product was generated from the default route using the controller name `ProductCatalog` and the product id. The link for adding the product to the cart was also from the default route, but used the controller name `Cart` instead. As you can see in **Figure 9**, the link for adding the product to the cart is associated to the response based on product availability (`product.IsAvailable`). The logic for providing links to the client will pretty much depend on the business rules typically enforced in the controllers.

Wrapping Up

Hypermedia is a powerful feature that allows clients and servers to evolve independently. By using links or other hypermedia artifacts such as forms offered by the server at different stages, clients can be successfully decoupled from the server business workflow that drives the interaction. ■

PABLO CIBRARO is an internationally recognized expert with more than 12 years of experience in designing and implementing large distributed systems with Microsoft technologies. He is a *Connected Systems MVP*. For the last nine years Cibraro has helped numerous Microsoft teams develop tools and frameworks for building service-oriented applications with Web Services, Windows Communication Foundation, ASP.NET and Windows Azure. He blogs at weblogs.asp.net/cibrax and you can follow him on Twitter at twitter.com/cibrax.

THANKS to the following technical expert for reviewing this article: *Daniel Roth*

Version Control in the TFS Client Object Model

Jeff Bramwell

This article is a follow-up to “Using the Team Foundation Server Client Object Model,” written by members of the Visual Studio ALM Rangers in the August 2012 issue (msdn.microsoft.com/magazine/jj553516). So far, we’ve introduced the Team Foundation Server (TFS) client object model, and now I’ll introduce the version control APIs.

To recap, the ALM Rangers are a group of experts who promote collaboration among the Visual Studio product group, Microsoft Services and the Microsoft Most Valuable Professional (MVP) community by addressing missing functionality, removing adoption blockers and publishing best practices and guidance based on real-world experiences.

This article discusses:

- Required assemblies and namespaces
- Interacting with the version control system
- Obtaining the latest source code from the repository
- Identifying files and folders for download
- Checking out code
- Checking in changes
- Retrieving the history of files or folders

Technologies discussed:

Team Foundation Server

Code download available at:

vsarguidance.codeplex.com/downloads/get/527603

If someone were to ask you to explain what TFS is, chances are you’d mention version control within the first few sentences. Although version control does play an important role within TFS, you can see in **Figure 1** that there’s much more to TFS. As with many features within TFS, the version control subsystem is accessible via the TFS object model. This accessibility provides you with an extensibility model that you can leverage within your own custom tools and processes.

Assemblies and Namespaces

Before you can access the functionality provided within the TFS object model, you must first understand the required assemblies and namespaces. You’ll recall the first article used the namespace `Microsoft.TeamFoundation.Client`. This namespace contains the

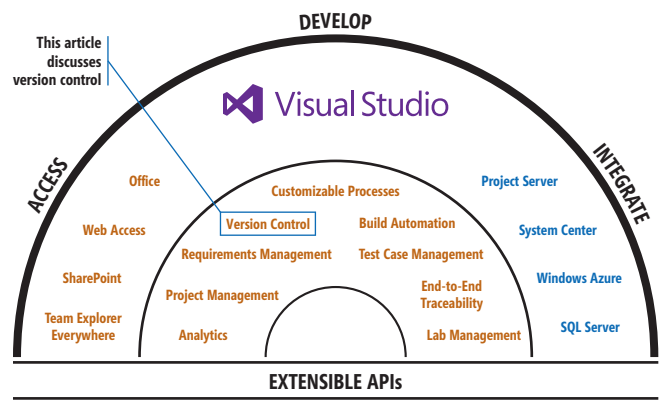


Figure 1 Team Foundation Server Features

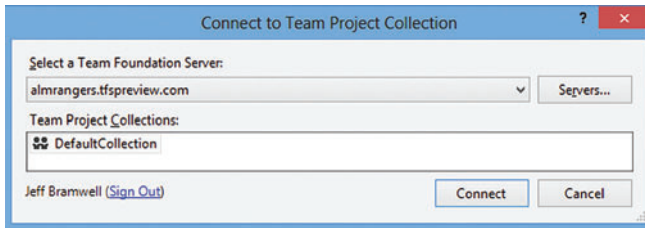


Figure 2 The TeamProjectPicker Dialog

classes and methods necessary for connecting to a TFS configuration server, and it's located within the identically named assembly. This namespace is central to all TFS object model-related development.

When working with version control, we must also utilize the namespace `Microsoft.TeamFoundation.VersionControl.Client`. This namespace contains the classes necessary for interacting with the TFS version control system. Utilizing the APIs within this namespace allows you to access files and folders, pending changes, merges, branches, and so on. The `VersionControlServer` class within this namespace is the main class that provides access to the TFS version control repository.

A Simple Example to Start

The `VersionControlServer` class exposes many properties, methods and events for interacting with version control within TFS. I'll start with a simple example: retrieving the latest changeset ID.

The three basic steps required to interact with most of the APIs exposed by the TFS object model are:

1. Connect to a TFS configuration server.
2. Obtain a reference to the TFS service you plan to utilize.
3. Make use of the various properties, methods and events provided by the service.

Taking a slightly different approach to connecting to TFS, as opposed to the examples presented in the August article, I'm going to connect to TFS using the `TeamProjectPicker` class. The `TeamProjectPicker` class displays a standard dialog for connecting to TFS servers. This class is not only useful for full-featured applications but is also very handy for simple utilities where you might need to switch among multiple instances of TFS.

Create a new instance of `TeamProjectPicker` and display it using the `ShowDialog` method:

```
private TfsTeamProjectCollection _tpc;
using (var picker = new TeamProjectPicker(TeamProjectPickerMode.NoProject, false))
{
    if (picker.ShowDialog() == DialogResult.OK)
    {
        _tpc = picker.SelectedTeamProjectCollection;
    }
}
```

This code will display a dialog similar to that shown in **Figure 2**.

Clicking `Connect` will return an instance of `TfsTeamProjectCollection` representing the selected Team Project Collection (TPC). If you prefer to use a more programmatic approach (that is, no user interaction) to connect to TFS, refer back to the August article for further examples.

Once you've obtained a reference to a `TfsTeamProjectCollection`, it can be used to obtain an instance of the `VersionControlServer` service:

```
var vcs = _tpc.GetService<VersionControlServer>();
```

Once you have a reference to the service you can make use of the methods exposed by the service:

```
var latestId = vcs.GetLatestChangesetId();
```

This is a simple example, but it does demonstrate the basic steps for interacting with the version control system in TFS. However, few applications are this simple.

"Getting Latest"

A common scenario related to version control is obtaining the latest source code from the repository—that is, "getting latest." While working within Visual Studio, you typically get the latest source code by right-clicking a file or folder within the Source Control Explorer (SCE) and selecting `Get Latest Version`. For this to work properly, you must also have a mapped workspace selected. When downloading the latest source code from the server, the selected workspace determines where it will be stored.

Follow these steps to programmatically obtain the latest source code:

1. Connect to a TFS configuration server.
2. Obtain a reference to the version control service.
3. Utilize an existing workspace or create a new, temporary workspace.
4. Map the workspace to a local folder.
5. Download the desired files from the workspace.

Building on the previous example, add the code shown in **Figure 3**.

If a workspace already exists and you'd like to use it, replace lines 1-4 in **Figure 3** with the following:

```
// Get a reference to an existing workspace,
// in this case, "DEMO_Workspace"
var workspace = vcs.GetWorkspace("DEMO_Workspace",
    _tpc.AuthorizedIdentity.UniqueName);
```

Note that if you don't know the name of the workspace or don't want to specify it, you can call `GetWorkspace` (see preceding code sample), passing in only a local path. This will return the workspace mapped to the local path.

Figure 3 Get Latest from Version Control

```
// Create a temporary workspace
var workspace = vcs.CreateWorkspace(Guid.NewGuid().ToString(),
    _tpc.AuthorizedIdentity.UniqueName,
    "Temporary workspace for file retrieval");
// For this workspace, map a server folder to a local folder
workspace.Map($"{Demo/TFS_VC_API}", @"C:\Dev\Test");
// Create an ItemSpec to determine which files and folders are retrieved
// Retrieve everything under the server folder
var fileRequest = new GetRequest(
    new ItemSpec($"{Demo/TFS_VC_API}", RecursionType.Full),
    VersionSpec.Latest);
// Get latest
var results = workspace.Get(fileRequest, GetOptions.GetAll | GetOptions.Overwrite);
```

Figure 4 VersionSpec Types

VersionSpec	Description
ChangesetVersionSpec	Specifies a version based on a changeset number.
DateVersionSpec	Specifies a version based on a date/time stamp.
LabelVersionSpec	Specifies a version based on a label.
LatestVersionSpec	Represents the latest valid version in the repository.
WorkspaceVersionSpec	Specifies a version based on a workspace name/owner.

Changeset	Change	User	Date	Path	Comment
231	edit	Jeff Bramwell	9/15/2012 4:05:53 PM	S:/Demo/T...	Adding history code.
230	edit	Jeff Bramwell	9/13/2012 10:30:28 PM	S:/Demo/T...	Another section down :-)
229	edit	Jeff Bramwell	9/4/2012 10:37:42 PM	S:/Demo/T...	Added Get Latest example.
226	edit	Jeff Bramwell	9/3/2012 10:45:19 PM	S:/Demo/T...	Adding simple VC example.
225	add	Jeff Bramwell	9/3/2012 10:26:51 PM	S:/Demo/T...	Initial check in.

Figure 5 History for Form1.cs

You don't need to map the workspace programmatically, so you can also remove lines 5 and 6.

Identifying Files and Folders for Download

As you might expect, several of the APIs provided by TFS allow you to query the version control server for specific items as well as specific versions of items. In the previous example, when I created a new instance of `GetRequest`, I had to provide an instance of `ItemSpec`. An `ItemSpec`, short for item specification, describes a set of files or folders. These items can exist on your local machine or in the version control server. In this specific example, I'm building an `ItemSpec` to return all files within the server folder `"$/Demo/TFS_VC_API"`.

The second parameter in the `ItemSpec` constructor used here specifies `RecursionType`, which can be `None`, `Full` or `OneLevel`. This value determines how many levels deep an API should consider when querying items. Specifying a `RecursionType` of `OneLevel` will query or return items from only the topmost level (relative to the `ItemSpec`). A value of `Full` will query or return items from the topmost level as well as all levels below (again, relative to the `ItemSpec`).

Whereas an `ItemSpec` determines which items to consider based on name and location when querying the version control system, a `VersionSpec`, short for version specification, provides the

Figure 6 Retrieve Item History

```

1. var vcs = _tpc.GetService<VersionControlServer>();
2. var results = vcs.QueryHistory(
3.     "$/Demo/TFS_VC_API/Form1.cs", // The item (file) to query history for
4.     VersionSpec.Latest,           // We want to query the latest version
5.     0,                            // We're not interested in the Deletion ID
6.     RecursionType.Full,           // Recurse all folders
7.     null,                         // Specify null to query for all users
8.     new ChangesetVersionSpec(1), // Starting version is the 1st changeset
                                   // in TFS
9.     VersionSpec.Latest,           // Ending version is the latest version
                                   // in TFS
10.    int.MaxValue,                 // Maximum number of entries to return
11.    true,                         // Include changes
12.    false);                       // Slot mode
13. if (results != null)
14. {
15.     foreach (var changeset in (IEnumerable<Changeset>)results)
16.     {
17.         if (changeset.Changes.Length > 0)
18.         {
19.             foreach (var change in changeset.Changes)
20.             {
21.                 ResultsTextBox.Text +=
22.                     string.Format("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\r\n",
23.                     change.Item.ChangesetId,
24.                     change.ChangeType,
25.                     changeset.CommitterDisplayName,
26.                     change.Item.CheckinDate,
27.                     change.Item.ServerItem,
28.                     changeset.Comment);
29.             }
30.         }
31.     }

```

ability to limit item sets based on version. `VersionSpec` is an abstract class so it can't be instantiated directly. TFS provides several implementations of `VersionSpec` that you can make use of when querying the version control system. **Figure 4** lists the various implementations of `VersionSpec` provided out of the box with TFS 2012.

Going back to the previous example of creating a `GetRequest`, I specified `VersionSpec.Latest` as my version specification. `VersionSpec.Latest` is simply a reference to a singleton instance of `LatestVersionSpec` provided just for convenience. To retrieve code based on a specific label, for example, create an instance of `LabelVersionSpec`:

```

var fileRequest = new GetRequest(
    new ItemSpec("$Demo/TFS_VC_API", RecursionType.Full),
    new LabelVersionSpec("MyLabel"));

```

Checking out Code

Now that you know how to identify and retrieve specific items from the version control server, let's look at how you can check out source code. In TFS terms, to check out an item is to *pend an edit* on that item. To pend an edit on items within a particular workspace, you call the `Workspace.PendEdit` method. The `PendEdit` method has nine overloads, all of which require a path or an array of paths as well as a few other optional parameters. One of the optional parameters is `RecursionType`, which works exactly as previously described for `ItemSpec`.

For example, to check out all C# (.cs) files, make this call:

```

// This example assumes we have obtained a reference
// to an existing workspace by following the previous examples
var results = workspace.PendEdit("$Demo/TFS_VC_API/*.cs", RecursionType.Full);

```

In this example, I'm requesting that TFS pend edits on all C# files (via the `*.cs` wildcard) beneath the server folder `"$/Demo/TFS_VC_API"`. Because I'm specifying a `RecursionType` of `Full`, I'll check out C# files in all folders beneath the specified path. The specific method signature used in this example will also download the checked-out files to the local path as mapped by the specified workspace. You can use one of the overloaded versions of this method that accepts an argument of type `PendChangesOptions` and specify `PendChangesOption.Silent` to suppress the downloading of files when pending edits. The value returned in `results` contains a count of items downloaded because of the call to `PendEdit`.

Edits aren't the only action that you can pend within the version control system. There are also methods for pending:

- Adds via `PendAdd`
- Branches via `PendBranch`
- Deletes via `PendDelete`
- Properties via `PendPropertyName`
- Renames via `PendRename`
- Undeletes via `PendUndelete`

For example, the following code pends a new branch, named `Dev`, from the folder `Main`:

```

// This example assumes we have obtained a reference
// to an existing workspace by following the previous examples
var results = workspace.PendBranch("$Demo/TFS_VC_API/Main",
    "$Demo/TFS_VC_API/Dev", VersionSpec.Latest);

```

We'll cover branching and merging using the APIs in more detail in a future article.

231	Edit	Jeff Bramwell	9/15/2012 4:05:53 PM	\$/Demo/TFS_VC_API/Main/Source/TFS_VC_API/TFS_VC_API/Form1.cs	Adding history code.
230	Edit	Jeff Bramwell	9/13/2012 10:30:28 PM	\$/Demo/TFS_VC_API/Main/Source/TFS_VC_API/TFS_VC_API/Form1.cs	Another section down :)
229	Edit	Jeff Bramwell	9/4/2012 10:37:42 PM	\$/Demo/TFS_VC_API/Main/Source/TFS_VC_API/TFS_VC_API/Form1.cs	Added Get Latest example.
226	Edit	Jeff Bramwell	9/3/2012 10:45:19 PM	\$/Demo/TFS_VC_API/Main/Source/TFS_VC_API/TFS_VC_API/Form1.cs	Adding simple VC example.
225	Add, Edit, Encoding	Jeff Bramwell	9/3/2012 10:26:51 PM	\$/Demo/TFS_VC_API/Main/Source/TFS_VC_API/TFS_VC_API/Form1.cs	Initial check in.

Figure 7 History from API

Checking in Changes

Once you've made changes to one or more of the checked-out files, you can check them back in via the `Workspace.CheckIn` method. Before you call the `CheckIn` method, however, you must first obtain a list of pending changes for the workspace by calling `Workspace.GetPendingChanges`. If you don't specify any parameters for the `GetPendingChanges` method, you'll get back all pending changes for the workspace. Otherwise, you can make use of one of the other 11 overloads and filter the list of pending changes returned by the call to TFS.

The following example will check in all pending changes for the workspace:

```
// This example assumes we have obtained a reference
// to an existing workspace by following the previous examples
var pendingChanges = workspace.GetPendingChanges();
var results = workspace.CheckIn(pendingChanges, "My check in.");
```

In the first line of code, I'm getting a list of all pending changes for the workspace. In the second line, I check everything back in to the version control server specifying a comment to be associated with the changeset. The value returned in `results` contains a count of the items checked in. If there's one or more pending changes and `results` comes back as zero, then no differences were found in the pending items between the server and the client.

Pending edits aren't the only changes checked in to the version control system. You also check in:

- Additions
- Branches
- Deletions/Undeletes
- Properties
- Renames

You can also undo pending changes by calling the `Workspace.Undo` method. As with the `CheckIn` method, you must also specify which pending changes you want to undo. The following example will undo all pending changes for the workspace:

```
var pendingChanges = workspace.GetPendingChanges();
var results = workspace.Undo(pendingChanges);
```

Retrieving History

A common task within Team Explorer is viewing the history of one or more files or folders. You might find the need to do this programmatically as well. As you might expect, there's an API for querying history. In fact, the method I'm going to discuss is available from the `VersionControlServer` instance (as represented by the variable `vcs` in the previous examples). The method is `VersionControlServer.QueryHistory`, and it has eight overloads. This method provides the capability of querying the version control server in many different ways, depending on the types and values of the parameters passed into the method call.

Figure 5 shows what the history view for the file `Form1.cs` might look like within the SCE.

You can replicate this functionality programmatically using the code shown in Figure 6.

Pay special attention to the argument on line 11 in Figure 6. I'm specifying a value of `true` for the parameter `includeChanges`. If you specify `false`, then specific changes for the version history won't be included in the returned results and the Figure 6 example won't display the details. You can still display basic changeset history without returning the changes, but some detail will be unavailable.

Running the Figure 6 example produces the results shown in Figure 7.

There are many other variations of calling the `QueryHistory` API. For ideas on how you might make use of this method, simply play around with the history features in the SCE. You can also query much more than history. For example, there are other query-related methods provided by the `VersionControlServer`, such as:

- `QueryBranchObjectOwnership`
- `QueryBranchObjects`
- `QueryLabels`
- `QueryMergeRelationships`
- `QueryMerges`
- `QueryMergesExtended`
- `QueryMergesWithDetails`
- `QueryPendingSets`
- `QueryRootBranchObjects`
- `QueryShelvedChanges`
- `QueryShelvesets`
- `QueryWorkspaces`

There's a lot of information available from the version control server, and the various query methods provide you with a window into that information.

Next Steps

The Rangers have only begun to touch on the version control features exposed by the TFS object model. The features covered in this article can be useful and powerful, but myriad features are exposed by the TFS version control server that we haven't covered. Some examples of these features include branching and merging, shelvesets, labels and version control events. With this series of articles, we hope to expose many of these APIs and provide you with the knowledge you need to make better use of the TFS object model. Stay tuned for more. ■

JEFF BRAMWELL is director of enterprise architecture at Farm Credit Services of America. He has more than 20 years of software development experience and always strives to follow the mantra of start simple and work your way up from there. His blog is at devmatter.blogspot.com. You can follow him on Twitter at twitter.com/jbramwell.

THANKS to the following technical experts for reviewing this article:
Brian Blackman, Mike Fourie and Willy-Peter Schaub



Artificial Immune Systems for Intrusion Detection

An artificial immune system (AIS) for intrusion detection is a software system that models some parts of the behavior of the human immune system to protect computer networks from viruses and similar cyber attacks. The essential idea is that the human immune system—which is a complex system consisting of lymphocytes (white blood cells), antibodies and many other components—has evolved over time to provide powerful protection against harmful toxins and other pathogens. So, modeling the behavior of the human immune system may provide an effective architecture against cyber attacks.

In this article I describe some of the principles of artificial immune systems and present a program to demonstrate these principles. Work on AIS protection is still relatively new and, in my opinion, no commercial implementations are quite ready for prime time. The code presented in this article will not directly enable you to create a realistic network-intrusion system, but there are at least four reasons why you might find this article worth reading. First, the code will give you a starting point for hands-on experimentation with a simple AIS system. Second, the principles explained will get you over the rather difficult initial hurdle to this area and allow you to understand research papers on AIS. Third, several of the programming techniques used in this article, in particular r-chunks bit matching and negative selection, can be useful in other programming scenarios. And fourth, you may just find the idea of modeling a software system based on the behavior of the human immune system interesting in its own right.

The best way to get a feel for where I'm headed is to take a look at the screenshot of a demo run, shown in **Figure 1**. The demo program begins by creating a set of six normal patterns (patterns that are known not to be part of some cyber attack) that represent TCP/IP network packets in binary form. This is called the self-set in AIS terminology. Of course, in a real AIS system, the self-set would likely contain tens or hundreds of thousands of patterns, and each pattern would be much larger (typically 48-256 bits) than the 12 bits used here. Next, the demo program creates three artificial lymphocytes. Each lymphocyte has a simulated antibody that has four bits (again, artificially small), an age and a stimulation field. The antibody field is essentially a detector. As you'll see shortly, the lymphocytes are created so that none of them

detect any of the patterns in the self-set. For now, observe that each lymphocyte has three consecutive 0s or 1s but none of the patterns in the self-set has three consecutive equal bit values.

After the system has been initialized, the demo program begins a tiny simulation with six input patterns. The first input is detected by lymphocyte 1, but because each lymphocyte has an activation threshold, the lymphocyte doesn't trigger an alarm. At time $t = 3$, lymphocyte 1 detects another suspicious input but, again, is not yet over the threshold. But at time $t = 5$, lymphocyte 1 detects a third suspicious input packet and a simulated alert is triggered.

In terms of an intrusion-detection system, antigens correspond to TCP/IP network packets whose contents contain some sort of harmful data, such as a computer virus.

In the sections that follow, I'll first explain the key concepts of the human immune system that are used to model an AIS. Then I'll walk you through the code that produced the screenshot in **Figure 1**. I'll conclude by giving you some additional references and a few opinions about AIS. This article assumes you have at least intermediate-level programming skills with a modern programming language. I use C# for the demo program, but I'll explain where you'll need to make changes if you want to refactor my code into another language such as Visual Basic .NET or Python. I present all the code that generated the screenshot in **Figure 1**; the code is also available at archive.msdn.microsoft.com/mag201301TestRun.

The Human Immune System

The key elements of a highly simplified immune system are illustrated in **Figure 2**. Harmful items are proteins called antigens. In **Figure 2** the antigens are colored red and have sharp corners. The human body also contains many non-harmful antigens called self-antigens, or self-items. These are naturally occurring proteins and in **Figure 2** are colored green and have rounded sides.

Code download available at archive.msdn.microsoft.com/mag201301TestRun.

Antigens are detected by lymphocytes. Each lymphocyte has several antibodies, which can be thought of as detectors. Each antibody is specific to a particular antigen. Typically, because antibody-antigen matching is only approximate, a lymphocyte will not trigger a reaction when a single antibody detects a single antigen. Only after several antibodies detect their corresponding antigens will a lymphocyte become stimulated and trigger some sort of defensive reaction.

Notice that no lymphocyte has antibodies that detect a self-item. Real antibodies are generated by the immune system in the thymus, but any antibodies that detect self-items are destroyed before being released into the blood stream, a process called apoptosis.

In terms of an intrusion-detection system, antigens correspond to TCP/IP network packets whose contents contain some sort of harmful data, such as a computer virus. Self-antigens correspond to normal, non-harmful network packets. An antibody corresponds to a bit pattern that approximately matches an unknown, potentially harmful network packet. A lymphocyte represents two or more antibodies/detectors. Apoptosis is modeled using a technique called negative selection.

Overall Program Structure

The demo program shown in **Figure 1** is a single C# console application named `ArtificialImmuneSystem`. I used Visual Studio 2010, but any version of Visual Studio that has the Microsoft .NET Framework 2.0 or later should work. I renamed the Visual Studio template-generated file named `Program.cs` to the more descriptive `ArtificialImmuneSystemProgram.cs` and renamed the corresponding class as well. The overall program structure is listed in **Figure 3**.

I deleted all the template-generated using statements except for references to the `System` and `System.Collections.Generic` namespaces. I added a reference to the `System.Collections` namespace so I could have access to the `BitArray` class. After a start-up message, I instantiated a static `Random` object using an arbitrary seed value of 1.

If you compare the code in the `Main` method in **Figure 3** with the screenshot in **Figure 1**, you shouldn't have too much trouble understanding the program logic. The key to the AIS demo is the definition of the class `Lymphocyte`. Note that in order to keep the size of the demo code small and the key ideas clear, I removed all the normal error checking you'd likely include during experimentation.

The Lymphocyte Class

The `Lymphocyte` class has four data fields:

```
public BitArray antibody; // Detector
public int[] searchTable; // For fast detection
public int age;           // Not used; could determine death
public int stimulation;   // Controls triggering
```

I declare all fields with public scope for simplicity. The antibody field is a `BitArray`. If you're not familiar with `BitArray`, the idea is that using a normal array of `int` to represent a collection of bits is highly inefficient because each `int` requires 32 bits of storage. A bit array condenses 32 bits of information into a single `int` of storage, plus some overhead for the class. Programming languages

```
file:///C:/ArtificialImmuneSystem/bin/Debug/ArtificialImmuneSystem.EXE
Begin Artificial Immune System for Intrusion Detection demo
Loading self-antigen set '<normal> historical patterns'
0: 1 0 0 1 0 1 1 0 1 0 0 1
1: 1 1 0 0 1 0 1 0 1 1 0 0
2: 1 0 1 1 0 0 1 1 0 1 0 1
3: 0 0 1 1 0 1 0 1 1 0 1 1
4: 0 1 0 1 0 1 0 0 1 1 0 1
5: 0 0 1 0 1 0 1 0 0 1 0 0

Creating lymphocyte set using negative selection and r-chunks detection
0: antibody = 0 0 0 1 age = 0 stimulation = 0
1: antibody = 0 1 1 1 age = 0 stimulation = 0
2: antibody = 1 0 0 0 age = 0 stimulation = 0

Begin AIS intrusion detection simulation
=====
Incoming pattern = 1 0 1 0 1 1 1 1 0 1 1
Incoming pattern not detected by lymphocyte 0
Incoming pattern detected by lymphocyte 1
Lymphocyte 1 not over stimulation threshold
Incoming pattern not detected by lymphocyte 2
=====
Incoming pattern = 1 0 1 1 0 1 1 0 0 1 1 0
Incoming pattern not detected by lymphocyte 0
Incoming pattern not detected by lymphocyte 1
Incoming pattern not detected by lymphocyte 2
=====
Incoming pattern = 0 0 1 0 0 1 0 0 1 0 1 0
Incoming pattern not detected by lymphocyte 0
Incoming pattern not detected by lymphocyte 1
Incoming pattern not detected by lymphocyte 2
=====
Incoming pattern = 1 0 0 0 0 0 1 0 1 1 1 0
Incoming pattern detected by lymphocyte 0
Lymphocyte 0 not over stimulation threshold
Incoming pattern detected by lymphocyte 1
Lymphocyte 1 not over stimulation threshold
Incoming pattern detected by lymphocyte 2
Lymphocyte 2 not over stimulation threshold
=====
Incoming pattern = 0 0 0 1 0 1 1 0 0 0 0 0
Incoming pattern detected by lymphocyte 0
Lymphocyte 0 not over stimulation threshold
Incoming pattern not detected by lymphocyte 1
Incoming pattern detected by lymphocyte 2
Lymphocyte 2 not over stimulation threshold
=====
Incoming pattern = 0 0 0 0 1 1 1 1 0 0 1 0
Incoming pattern detected by lymphocyte 0
Lymphocyte 0 stimulated! Check incoming as possible intrusion!
Incoming pattern detected by lymphocyte 1
Lymphocyte 1 stimulated! Check incoming as possible intrusion!
Incoming pattern not detected by lymphocyte 2
=====
End AIS IDS demo
```

Figure 1 Artificial Immune System Demo

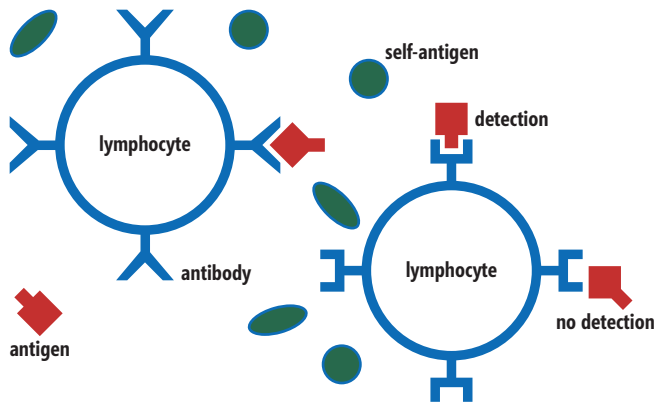


Figure 2 Key Elements of a Simplified Immune System

that don't have a BitArray-like class require you to do low-level bit manipulation with bit masking and bit operations.

The searchTable field is an array that's used by the Detects method to greatly increase performance. The age field isn't used in my demo program; many AIS systems track the age of a simulated lymphocyte and probabilistically kill off and generate new lymphocytes

based on age. The stimulation field is a counter that tracks how many times a Lymphocyte object has detected a possible threat. In this demo, when a lymphocyte stimulation value exceeds the stimulationThreshold value of 3, an alert is triggered.

The Lymphocyte constructor is:

```
public Lymphocyte(BitArray antibody)
{
    this.antibody = new BitArray(antibody);
    this.searchTable = BuildTable();
    this.age = 0;
    this.stimulation = 0;
}
```

The constructor accepts a single BitArray parameter that represents an antigen/detector. The searchTable array is instantiated using a private helper method named BuildTable, which I'll present shortly.

One of the key parts of any AIS system is the routine that determines whether an antigen detects a pattern. Requiring an exact match isn't feasible (and doesn't mimic real antigen behavior). Early work on AIS used a technique called r-contiguous bits matching, in which both an antigen and an input pattern have the same number of bits and detection occurs when antigen and pattern match in r-consecutive bits. Later research indicated that a better detection algorithm is r-chunks bit matching. The r-chunks bit matching is similar to r-contiguous bits

Figure 3 Artificial Immune System Program Structure

```
using System;
using System.Collections.Generic;
using System.Collections; // for BitArray

namespace ArtificialImmuneSystem
{
    class ArtificialImmuneSystemProgram
    {
        static Random random;

        static void Main(string[] args)
        {
            Console.WriteLine("\nBegin Artificial Immune System for Intrusion" +
                " Detection demo\n");

            random = new Random(1);

            int numPatternBits = 12;
            int numAntibodyBits = 4;
            int numLymphocytes = 3;
            int stimulationThreshold = 3;

            Console.WriteLine("Loading self-antigen set ('normal' historical patterns)");
            List<BitArray> selfSet = LoadSelfSet(null);
            ShowSelfSet(selfSet);

            Console.WriteLine("\nCreating lymphocyte set using negative selection" +
                " and r-chunks detection");
            List<Lymphocyte> lymphocyteSet = CreateLymphocyteSet(selfSet, numAntibodyBits,
                numLymphocytes);
            ShowLymphocyteSet(lymphocyteSet);

            Console.WriteLine("\nBegin AIS intrusion detection simulation\n");
            int time = 0;
            int maxTime = 6;
            while (time < maxTime)
            {
                Console.WriteLine("=====");
                BitArray incoming = RandomBitArray(numPatternBits);
                Console.WriteLine("Incoming pattern = " +
                    BitArrayAsString(incoming) + "\n");

                for (int i = 0; i < lymphocyteSet.Count; ++i)
                {
                    if (lymphocyteSet[i].Detects(incoming) == true)
                    {
                        Console.WriteLine("Incoming pattern detected by lymphocyte " + i);

                        ++lymphocyteSet[i].stimulation;
                        if (lymphocyteSet[i].stimulation >= stimulationThreshold)
                        {
                            Console.WriteLine("Lymphocyte " + i + " stimulated!" +
                                " Check incoming as possible intrusion!");
                        }
                        else
                        {
                            Console.WriteLine("Lymphocyte " + i + " not over stimulation" +
                                " threshold");
                        }
                    }
                    else
                    {
                        Console.WriteLine("Incoming pattern not detected by lymphocyte " + i);
                    }
                }
                ++time;
                Console.WriteLine("=====");
            } // Simulation loop

            Console.WriteLine("\nEnd AIS IDS demo\n");
            Console.ReadLine();
        } // Main

        public static List<BitArray> LoadSelfSet(string dataSource) {..}
        public static void ShowSelfSet(List<BitArray> selfSet) {..}
        public static string BitArrayAsString(BitArray ba) {..}
        public static List<Lymphocyte> CreateLymphocyteSet(List<BitArray> selfSet,
            int numAntibodyBits, int numLymphocytes) {..}
        private static bool DetectsAny(List<BitArray>
            selfSet, Lymphocyte lymphocyte) {..}
        public static void ShowLymphocyteSet(List<Lymphocyte> lymphocyteSet) {..}
        public static BitArray RandomBitArray(int numBits) {..}
    } // Program

    public class Lymphocyte
    {
        public BitArray antibody; // detector
        public int[] searchTable; // for fast detection
        public int age; // not used; could determine death
        public int stimulation; // controls triggering

        public Lymphocyte(BitArray antibody) {..}
        private int[] BuildTable() {..}
        public bool Detects(BitArray pattern) {..}
        public override int GetHashCode() {..}
        public override string ToString() {..}
    }
} // ns
```

Figure 4 The Detects Method

```
public bool Detects(BitArray pattern) // Adapted KMP algorithm
{
    int m = 0;
    int i = 0;
    while (m + i < pattern.Length)
    {
        if (this.antibody[i] == pattern[m + i])
        {
            if (i == antibody.Length - 1)
                return true;
            ++i;
        }
        else
        {
            m = m + i - this.searchTable[i];
            if (searchTable[i] > -1)
                i = searchTable[i];
            else
                i = 0;
        }
    }
    return false; // Not found
}
```

matching except that the antigen detector is smaller than the pattern to check, and detection occurs when the antigen matches any subset of the pattern. For example, if an antigen is 110 and a pattern is 000110111, the antigen detects the pattern starting at index 3.

If you think about r-chunks matching for a moment, you'll realize that it's almost the same as a string substring function. The only difference is that r-chunks matching matches bits and substring matches characters.

In the preceding example, a simple approach to r-chunks matching would be to examine the pattern starting at index 0, then at index 1, then 2 and so on. However, this approach is very slow in most situations. There are several sophisticated substring algorithms that preprocess the smaller detector string to create an array (usually called a table). This search table can be used to skip ahead when a mismatch is encountered and greatly improve performance. In situations where the smaller detector string is repeatedly used to check different patterns—as in AIS intrusion detection—the time and memory needed to create the search table is a small price to pay for dramatically improved performance.

Dealing with false positives—that is, triggering an alert on a non-harmful input pattern—is a major challenge for AIS systems.

The Lymphocyte class Detects method uses the Knuth-Morris-Pratt substring algorithm applied to a BitArray. The Detects method accepts an input pattern such as 000110111 and returns true if the current object's antigen, such as 110, matches the pattern. The code for the Detects method is listed in **Figure 4**.

The Detects method assumes the existence of the search table. Recall that the Lymphocyte constructor calls a helper method

BuildTable to create the searchTable field. The code for BuildTable is listed in **Figure 5**.

The Lymphocyte class defines overridden GetHashCode and ToString methods. The GetHashCode method is used to prevent duplicate Lymphocyte objects and is:

```
public override int GetHashCode()
{
    int[] singleInt = new int[1];
    antibody.CopyTo(singleInt, 0);
    return singleInt[0];
}
```

This implementation assumes that the BitArray antibody field has 32 bits or less. In realistic situations where the antibody field has more than 32 bits, dealing with duplicate Lymphocyte objects is not so simple. One approach would be to define a custom hash code method that returns a BigInteger type (available in the .NET Framework 4 and later).

The Lymphocyte ToString method used in the demo program is:

```
public override string ToString()
{
    string s = "antibody = ";
    for (int i = 0; i < antibody.Length; ++i)
        s += (antibody[i] == true) ? "1 " : "0 ";
    s += " age = " + age;
    s += " stimulation = " + stimulation;
    return s;
}
```

Creating the Self-Set

You are certainly familiar with standard (non-AIS) computer virus-detection software such as Microsoft Security Essentials. These systems work by storing a local database of known computer virus definitions. When a known virus pattern is detected, an immediate alert is triggered. Such antivirus systems have trouble dealing with variations on existing viruses, and fail entirely in most situations when faced with a completely new virus. AIS intrusion detection works in the opposite way by maintaining a set of input patterns that are known to be non-harmful and then triggering an alert when an unknown pattern is detected. This allows AIS intrusion-detection systems—in principle, at least—to detect new viruses. However, dealing with false positives—that is, triggering an alert on a non-harmful input pattern—is a major challenge for AIS systems.

A real AIS intrusion-detection system would collect many thousands of normal input patterns over the course of days or weeks. These normal self-set patterns would be stored either on a local host or a server. Also, a real AIS system would continuously update the self-set (and the induced set of lymphocytes) of normal patterns to account for normal changes in network traffic over time. The demo program in this article creates an artificial, static self-set using method LoadSelfSet:

```
public static List<BitArray> LoadSelfSet(string dataSource)
{
    List<BitArray> result = new List<BitArray>();
    bool[] self0 = new bool[] { true, false, false, true, false, true,
                                true, false, true, false, false, true };
    // Etc.
    bool[] self5 = new bool[] { false, false, true, false, true, false,
                                true, false, false, true, false, false };
    result.Add(new BitArray(self0));
    // Etc.
    result.Add(new BitArray(self5));
    return result;
}
```

The method accepts a dummy not-used dataSource parameter to suggest that in a realistic scenario the self-set data would not

be hardcoded. The BitArray constructor, somewhat surprisingly, accepts an array of bool values where true represents a 1 bit and false represents a 0 bit. Observe that I generated the self-set data in such a way that no self-item has more than two consecutive 0s or 1s.

The demo program uses utility method ShowSelfSet, which calls helper method BitArrayAsString, to display the self-set:

```
public static void ShowSelfSet(List<BitArray> selfSet)
{
    for (int i = 0; i < selfSet.Count; ++i)
        Console.WriteLine(i + ": " + BitArrayAsString(selfSet[i]));
}

public static string BitArrayAsString(BitArray ba)
{
    string s = "";
    for (int i = 0; i < ba.Length; ++i)
        s += (ba[i] == true) ? "1 " : "0 ";
    return s;
}
```

Creating the Lymphocyte Set

If you refer back to the explanation of how the human immune system works, you'll note that the lymphocyte set should contain only Lymphocyte objects that do not detect any patterns in the self-set. Method CreateLymphocyteSet is listed in **Figure 6**.

Figure 5 The BuildTable Method

```
private int[] BuildTable()
{
    int[] result = new int[antibody.Length];
    int pos = 2;
    int cnd = 0;
    result[0] = -1;
    result[1] = 0;
    while (pos < antibody.Length)
    {
        if (antibody[pos - 1] == antibody[cnd])
        {
            ++cnd; result[pos] = cnd; ++pos;
        }
        else if (cnd > 0)
            cnd = result[cnd];
        else
        {
            result[pos] = 0; ++pos;
        }
    }
    return result;
}
```

Figure 6 The CreateLymphocyteSet Method

```
public static List<Lymphocyte> CreateLymphocyteSet(List<BitArray> selfSet,
    int numAntibodyBits, int numLymphocytes)
{
    List<Lymphocyte> result = new List<Lymphocyte>();
    Dictionary<int, bool> contents = new Dictionary<int, bool>();

    while (result.Count < numLymphocytes)
    {
        BitArray antibody = RandomBitArray(numAntibodyBits);
        Lymphocyte lymphocyte = new Lymphocyte(antibody);
        int hash = lymphocyte.GetHashCode();

        if (DetectsAny(selfSet, lymphocyte) == false &&
            contents.ContainsKey(hash) == false)
        {
            result.Add(lymphocyte);
            contents.Add(hash, true);
        }
    }
    return result;
}
```

In AIS terminology, method CreateLymphocyteSet uses negative selection. A random Lymphocyte object is generated and then tested to see if it does *not* detect any patterns in the self-set, and also that the lymphocyte is not already in the result set. This approach is rather crude, and there are other approaches that are more efficient. I use a Dictionary collection with a dummy bool value to track existing Lymphocyte objects; the HashSet in the .NET Framework 4.5 and later is a more efficient alternative.

A random Lymphocyte object is created by generating a random BitArray:

```
public static BitArray RandomBitArray(int numBits)
{
    bool[] bools = new bool[numBits];
    for (int i = 0; i < numBits; ++i)
    {
        int b = random.Next(0, 2); // between [0,1] inclusive
        bools[i] = (b == 0) ? false : true;
    }
    return new BitArray(bools);
}
```

Helper method DetectsAny accepts a self-set, and a lymphocyte scans through a self-set and returns true if any pattern in the self-set is detected by the antigen in the lymphocyte:

```
private static bool DetectsAny(List<BitArray> selfSet,
    Lymphocyte lymphocyte)
{
    for (int i = 0; i < selfSet.Count; ++i)
        if (lymphocyte.Detects(selfSet[i]) == true) return true;
    return false;
}
```

The demo program uses a utility method ShowLymphocyteSet to display the generated Lymphocyte objects:

```
public static void ShowLymphocyteSet(List<Lymphocyte> lymphocyteSet)
{
    for (int i = 0; i < lymphocyteSet.Count; ++i)
        Console.WriteLine(i + ": " + lymphocyteSet[i].ToString());
}
```

Wrapping Up

The code and explanations I've presented in this article should give you a solid basis for hands-on experimentation with an AIS. Researchers have suggested many options. For example, the demo program in this article fires an alert when a single lymphocyte detects unknown input patterns more than some threshold number of times. The idea here is that real pathogens emit many antigens. Another possibility is for the AIS system to trigger an alert only after multiple different lymphocytes detect the same unknown pattern.

It's important to point out that an AIS is not intended to be a single solution for intrusion detection. Rather, it's meant to be part of a multilayer defense that includes traditional antivirus software. Additionally, because work with an AIS is still relatively young, there are many unanswered questions. If you wish to examine some of the research on AIS, I recommend searching online for articles by authors S. Forrest, P. Williams and U. Aickelin. ■

Dr. James McCaffrey works for Volt Information Sciences Inc., where he manages technical training for software engineers working at the Microsoft Redmond, Wash., campus. He has worked on several Microsoft products including Internet Explorer and MSN Search. McCaffrey is the author of *“.NET Test Automation Recipes”* (Apress, 2006). He can be reached at jammc@microsoft.com.

THANKS to the following technical expert for reviewing this article:
Dan Liebling

HTML5+jQuery

Any App - Any Browser - Any Platform - Any Device



IGNITEUITM
INFRAGISTICS JQUERY CONTROLS



Download Your **Free Trial!**
www.infragistics.com/igniteui-trial



Infragistics Sales US 800 231 8588 • Europe +44 (0) 800 298 9055 • India +91 80 4151 8042 • APAC +61 3 9982 4545

Copyright 1996-2013 Infragistics, Inc. All rights reserved. Infragistics and Infragistics are registered trademarks of Infragistics, Inc.
The Infragistics logo is a trademark of Infragistics, Inc. All other trademarks or registered trademarks are the respective property of their owners.



.NET Collections: Getting Started with C5

I have a confession to make.

By day, I work as a mild-mannered .NET developer for Neudesic LLC, a consulting firm. But by night, after the wife and my sons are asleep, I sneak out of the house, laptop in hand, and journey to my secret hideout (a Denny's on 148th), and I... I write Java code.

Yes, folks, I live a double life as both a .NET developer and a Java—or, more accurately, a Java Virtual Machine (JVM)—developer. One of the interesting perks of living such a dual lifestyle is that I can see areas where the Microsoft .NET Framework has some great ideas that can be brought back to the JVM. One such idea was that of custom attributes, which the JVM adopted in Java5 (back in 2005 or so) with the name “annotations.” But the reverse is also true: The JVM did, in fact, get a few things right that the CLR and the .NET Base Class Library (BCL) didn't (or at least didn't get quite as right, if you're feeling a little defensive). One such area lies at the heart of the .NET BCL: collections.

Collections: A Critique

Part of the flaw in .NET collections lies in the fact that the BCL team had to write the silly things twice: once for the .NET Framework 1.0/1.1 release, before generics were available, and again for the .NET Framework 2.0, after generics were part of the CLR, because collections without strongly typed versions are just kind of silly. This automatically meant that one of these was bound to be left to rot, essentially, in favor of whatever enhancements or additions to the library were to come along. (Java ducked this particular problem by essentially “replacing” the non-genericized versions with genericized versions, which was only possible because of the way Java did generics—which isn't something I'm going to get into

here.) And, aside from the enhancements that came via LINQ in Visual Studio 2008 and C# 3.0, the collections library never really got much love after the 2.0 release, which itself more or less just re-implemented the System.Collections classes into a new namespace (System.Collections.Generic, or SCG) of strongly typed versions.

More importantly, though, the design of the .NET collections seems to have been more focused on getting something practical and useful out into the wild as part of the 1.0 release, rather than trying to think deeply about the design of collections and how they might be extended. This was one area in which the .NET Framework really (I suspect unintentionally) paralleled the Java world. When Java 1.0 shipped, it included a set of basic, utilitarian collections. But they had a few design flaws (the most egregious of which was the decision that had the Stack class, a last-in-first-out collection, directly extend the Vector class, which was basically an ArrayList). After Java 1.1 shipped, a few engineers at Sun Microsystems worked hard on a rewrite of the collections classes—which became known as the Java Collections—and shipped it as a part of Java 1.2.

Anyway, the .NET Framework is past due for a revamp of its collection classes, ideally in a way that's at least mostly compatible with the existing SCG classes. Fortunately, researchers at the IT University of Copenhagen in Denmark have produced a worthy successor and complement to the SCG classes: a library they call the Copenhagen Comprehensive Collection Classes for C#, or C5 for short.

C5 Logistics

To begin, C5 is found on the Web at itu.dk/research/c5 if you want to see the version history or get a link to a book (PDF) on C5, though the book is a few releases old. Or, alternatively, C5 is available through NuGet through the (by-now ubiquitous) Install-Package command, simply by typing “Install-Package C5.” Note that C5 is written to be available for both Visual Studio and Mono, and when NuGet installs the package, it will add references to both the C5.dll assembly as well as the C5Mono.dll assembly. These are redundant to each other, so delete the one you don't want. To explore C5 collections through a series of exploration tests, I created a Visual C# Test Project and added C5 to that project. Beyond that, the only notable change to the code is two “using” statements, which the C5 documentation also assumes:

```
using SCG = System.Collections.Generic;  
using C5;
```

The reason for the alias is simple: C5 “re-implements” a few interfaces and classes that are named the same in the SCG version, so aliasing the old stuff leaves it available to us but under a very short prefix (IList<T> is the C5 version, for example, whereas SCG.IList<T> is the “classic” version from SCG).

Figure 1 Getting Started with C5

```
// These are C5 IList and ArrayList, not SCG  
IList<String> names = new ArrayList<String>();  
names.AddAll(new String[] { "Hoover", "Roosevelt", "Truman",  
    "Eisenhower", "Kennedy" });  
  
// Print list:  
Assert.AreEqual("[ 0:Hoover, 1:Roosevelt, 2:Truman, 3:Eisenhower," +  
    " 4:Kennedy ]", names.ToString());  
  
// Print item 1 ("Roosevelt") in the list  
Assert.AreEqual("Roosevelt", names[1]);  
Console.WriteLine(names[1]);  
  
// Create a list view comprising post-WW2 presidents  
IList<String> postWWII = names.View(2, 3);  
// Print item 2 ("Kennedy") in the view  
Assert.AreEqual("Kennedy", postWWII[2]);  
  
// Enumerate and print the list view in reverse chronological order  
Assert.AreEqual("{ Kennedy, Eisenhower, Truman }",  
    postWWII.Backwards().ToString());
```


By the way, in case the lawyers ask, C5 is open sourced under an MIT license, so you're far more able to modify or enhance some of the C5 classes than you would be under a GNU General Public License (GPL) or GNU Lesser General Public License (LGPL).

C5 Design Overview

Looking at the C5 design approach, it seems similar to the SCG style, in that collections are broken into two “levels”: an interface layer that describes the interface and behavior expected of a given collection, and an implementation layer that provides the actual backing code for the one or more interfaces desired. The SCG classes approach this idea, but in some cases they don't follow through on it particularly well—for example, we don't have any flexibility in terms of the implementation of the `SortedSet<T>` (meaning, the choice of array-based, linked-list-based or hash-based, each of which has different characteristics regarding performance of insertion, traversal and so on). In some cases the SCG classes are simply missing certain collection types—a circular queue, for example (in which, when the last item in the queue is traversed, the iteration “wraps around” to the front of the queue again), or a simple “bag” collection (which offers no functionality except to contain items, thus avoiding unnecessary overhead of sorting, indexing and so on).

Granted, to the average .NET developer, this doesn't seem like much of a loss. But in many applications, as performance begins to become a central focus, choosing the right collection class to match the problem at hand becomes more critical. Is this a collection that will be established once and traversed frequently? Or is this a collection that will be added or removed frequently but traversed rarely? If this collection is at the heart of an application feature (or the app itself), the difference between these two could mean the difference between “Wow, this app screams!” and “Well, the users liked it, but just thought it was too slow.”

C5, then, holds as one of its core principles that developers should “code to interfaces, not implementations,” and, as such, offers more than a dozen different interfaces that describe what the underlying collection must provide. `ICollection<T>` is the base of it all, guaranteeing basic collection behavior, but from there we find `IList<T>`, `IIndexed<T>`, `ISorted<T>` and `ISequenceable<T>`, just to start. Here's the full list of interfaces, their relationships to other interfaces and their overall guarantees:

- An `SCG.IEnumerable<T>` can have its items enumerated. All collections and dictionaries are enumerable.
- An `IDirectedEnumerable<T>` is an enumerable that can be reversed, giving a backward enumerable that enumerates its items in the opposite order.
- An `ICollectionValue<T>` is a collection value. It doesn't support modification, is enumerable, knows how many items it has and can copy them to an array.
- An `IDirectedCollectionValue<T>` is a collection value that can be reversed into a backward collection value.
- An `IExtensible<T>` is a collection to which items can be added.
- An `IPriorityQueue<T>` is an extensible whose least and greatest items can be found (and removed) efficiently.
- An `ICollection<T>` is an extensible from which items can also be removed.

- An `ISequenceable<T>` is a collection whose items appear in a particular sequence (determined either by insertion order or item ordering).
- An `IIndexed<T>` is a sequenced collection whose items are accessible by index.
- An `ISorted<T>` is a sequenced collection in which items appear in increasing order; item comparisons determine the item sequence. It can efficiently find the predecessor or successor (in the collection) of a given item.
- An `IIndexedSorted<T>` is an indexed and sorted collection. It can efficiently determine how many items are greater than or equal to a given item *x*.
- An `IPersistentSorted<T>` is a sorted collection of which one can efficiently make a snapshot—that is, a read-only copy that will remain unaffected by updates to the original collection.
- An `IQueue<T>` is a first-in-first-out (FIFO) queue that in addition supports indexing.
- An `IStack<T>` is a last-in-first-out (LIFO) stack.
- An `IList<T>` is an indexed and therefore sequenced collection, where item order is determined by insertions and deletions. It derives from `SCG.IList<T>`.

In terms of the implementations, C5 has quite a few, including circular queues; array-backed as well as linked-list-backed lists, but also hashed-array lists and hashed-linked lists; wrapped arrays; sorted arrays; tree-based sets and bags; and more.

C5 Coding Style

Fortunately, C5 doesn't require a significant shift in coding style—and, even better, it supports all of the LINQ operations (because it builds on top of the SCG interfaces, of which the LINQ extension methods key off), so in some cases you can drop in a C5 collection at construction time without changing any of the code around it. See **Figure 1** for an example of this.

Even without having ever looked at the C5 documentation, it's pretty easy to figure out what's happening in these examples.

C5 vs. SCG Collection Implementations

This is just the tip of the iceberg with respect to C5. In my next column I'll look at some practical examples of using C5 instead of the SCG collection implementations—and some of the benefits that doing so gives us. I encourage you not to wait, though: Do the NuGet thing, pull C5 down and start exploring on your own—there's plenty there for you to entertain yourself in the meantime.

Happy coding! ■

TED NEWARD is an architectural consultant with Neudesic LLC. He has written more than 100 articles and is an author and coauthor of a dozen books, including “Professional F# 2.0” (Wrox, 2010). He's an F# MVP and noted Java expert, and speaks at both Java and .NET conferences around the world. He consults and mentors regularly—reach him at ted@tedneward.com or Ted.Neward@neudesic.com if you're interested in having him come work with your team. He blogs at blogs.tedneward.com and can be followed on Twitter at twitter.com/tedneward.

THANKS to the following technical expert for reviewing this article:
Immo Landwerth



Windows 8 Sound Generation with XAudio2

A Windows Store app for Windows 8 can play MP3 or WMA sound files easily using `MediaElement`—you simply give it a URI or a stream to the sound file. Windows Store apps can also access the Play To API for streaming video or audio to external devices.

But what if you need more sophisticated audio processing? Perhaps you'd like to modify the contents of an audio file on its way to the hardware, or generate sounds dynamically.

A Windows Store app can also perform these jobs through DirectX. Windows 8 supports two DirectX components for sound generation and processing—Core Audio and XAudio2. In the grand scheme of things, both of these are rather low-level interfaces, but Core Audio is lower than XAudio2 and is geared more toward applications that require a closer connection with audio hardware.

XAudio2 is the successor to DirectSound and the Xbox XAudio library, and it's probably your best bet for Windows Store apps that need to do interesting things with sound. While XAudio2 is intended primarily for games, that shouldn't stop us from using it for more serious purposes—such as making music or entertaining the business user with funny sounds.

XAudio2 version 2.8 is a built-in component of Windows 8. Like the rest of DirectX, the XAudio2 programming interface is based on COM. While it's theoretically possible to access XAudio2 from any programming language supported by Windows 8, the most natural and easiest language for XAudio2 is C++. Working with sound often requires high-performance code, so C++ is a good choice in that respect as well.

A First XAudio2 Program

Let's begin writing a program that uses XAudio2 to play a simple 5-second sound in response to the push of a button. Because you might be new to Windows 8 and DirectX programming, I'll take it a little slow.

I'll assume you have a version of Visual Studio installed that's suitable for creating Windows Store apps. In the New Project dialog box, select Visual C++ and Windows Store at the left, and Blank App (XAML) in the list of available templates. I gave my project the name `SimpleAudio`, and you can find that project among the downloadable code for this article.

In building an executable that uses XAudio2, you'll need to link the program with the `xaudio2.lib` import library. Bring up the project Properties dialog box by selecting the last item on the Project menu, or by right-clicking the project name in the Solution Explorer and selecting Properties. In the left column, select Configuration Properties and then Linker and Input. Click Additional

Dependencies (the top item) and the little arrow. Select Edit and type `xaudio2.lib` into the box.

You'll also want a reference to the `xaudio2.h` header file, so add the following statement to the precompiled headers list in `pch.h`:

```
#include <xaudio2.h>
```

In the `MainPage.xaml` file, I added a `TextBlock` for displaying any errors that the program might encounter working with the XAudio2 API, and a `Button` for playing a sound. These are shown in **Figure 1**.

The bulk of the `MainPage.xaml.h` header file is shown in **Figure 2**. I've removed the declaration of the `OnNavigatedTo` method because I won't be using it. The Click handler for the `Button` is declared, as are four fields connected with the program's use of XAudio2.

Any program that wishes to use XAudio2 must create an object that implements the `IXAudio2` interface. (You'll see how this is done shortly.) `IXAudio2` derives from the famous `IUnknown` class in COM, and it's inherently reference-counted, which means that it deletes its own resources when it's no longer referenced by a program. The `ComPtr` (COM Pointer) class in the `Microsoft::WRL` namespace turns a pointer to a COM object into a "smart pointer" that keeps track of its own references. This is the recommended approach for working with COM objects in a Windows Store app.

Any non-trivial XAudio2 program also needs pointers to objects that implement the `IXAudio2MasteringVoice` and `IXAudio2SourceVoice` interfaces. In XAudio2 parlance, a "voice" is an object that generates or modifies audio data. The mastering voice is conceptually a sound mixer that assembles all the individual voices and prepares them for the sound-generation hardware. You'll only have one of these, but you might have a number of source voices that generate separate sounds. (There are also ways to apply filters or effects to source voices.)

The `IXAudio2MasterVoice` and `IXAudio2SourceVoice` pointers are *not* reference-counted; their lifetimes are governed by the `IXAudio2` object.

Figure 1 The `MainPage.xaml` File for `SimpleAudio`

```
<Page x:Class="SimpleAudio.MainPage" ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Name="errorText"
      FontSize="24"
      TextWrapping="Wrap"
      HorizontalAlignment="Center"
      VerticalAlignment="Center" />

    <Button Name="submitButton"
      Content="Submit Audio Button"
      Visibility="Collapsed"
      HorizontalAlignment="Center"
      VerticalAlignment="Center"
      Click="OnSubmitButtonClick" />

  </Grid>
</Page>
```

Code download available at archive.msdn.microsoft.com/mag201301DXF.

Figure 2 The MainPage.xaml.h Header File for SimpleAudio

```
namespace SimpleAudio
{
    public ref class MainPage sealed
    {
    private:
        Microsoft::WRL::ComPtr<IXAudio2> pXAudio2;
        IXAudio2MasteringVoice * pMasteringVoice;
        IXAudio2SourceVoice * pSourceVoice;
        byte soundData[2 * 5 * 44100];

    public:
        MainPage();

    private:
        void OnSubmitButtonClick(Platform::Object^ sender,
            Windows::UI::Xaml::RoutedEventArgs^ args);
    };
}
```

I've also included a large array for 5 seconds worth of sound data:

```
byte soundData[5 * 2 * 44100];
```

In a real program, you'll want to allocate an array of this size at run time—and get rid of it when you don't need it—but you'll see shortly why I did it this way.

How did I calculate that array size? Although XAudio2 supports compressed audio, most programs that generate sound will stick with the format known as pulse-code modulation, or PCM.

Sound waveforms in PCM are represented by values of a fixed size at a fixed sampling rate. For music on compact disks, the sampling rate is 44,100 times per second, with 2 bytes per sample in stereo, for a total of 176,400 bytes of data for 1 second of audio. (When embedding sounds in an application, compression is recommended. XAudio2 supports ADPCM; WMA and MP3 are also supported in the Media Foundation engine.)

For this program, I've also chosen to use a sampling rate of 44,100 with 2 bytes per sample. In C++, each sample is therefore a short. I'll stick with monaural sound for now, so 88,200 bytes are required per second of audio. In the array allocation, that's multiplied by 5 for 5 seconds.

Creating the Objects

Much of the MainPage.xaml.cpp file is shown in **Figure 3**. All of the XAudio2 initialization is performed in the MainPage constructor. It begins with a call to XAudio2Create to obtain a pointer to an object that implements the IXAudio2 interface. This is the first step in using XAudio2. Unlike some COM interfaces, no call to CoCreateInstance is required.

Once the IXAudio2 object is created, the CreateMasteringVoice and CreateSourceVoice methods obtain pointers to the other two interfaces defined as fields in the header file.

Figure 3 MainPage.xaml.cpp

```
MainPage::MainPage()
{
    InitializeComponent();

    // Create an IXAudio2 object
    HRESULT hr = XAudio2Create(&pXAudio2);

    if (FAILED(hr))
    {
        errorText->Text = "XAudio2Create failure: " + hr.ToString();
        return;
    }

    // Create a mastering voice
    hr = pXAudio2->CreateMasteringVoice(&pMasteringVoice);

    if (FAILED(hr))
    {
        errorText->Text = "CreateMasteringVoice failure: " + hr.ToString();
        return;
    }

    // Create a source voice
    WAVEFORMATEX waveformat;
    waveformat.wFormatTag = WAVE_FORMAT_PCM;
    waveformat.nChannels = 1;
    waveformat.nSamplesPerSec = 44100;
    waveformat.nAvgBytesPerSec = 44100 * 2;
    waveformat.nBlockAlign = 2;
    waveformat.wBitsPerSample = 16;
    waveformat.cbSize = 0;

    hr = pXAudio2->CreateSourceVoice(&pSourceVoice, &waveformat);

    if (FAILED(hr))
    {
        errorText->Text = "CreateSourceVoice failure: " + hr.ToString();
        return;
    }

    // Start the source voice
    hr = pSourceVoice->Start();

    if (FAILED(hr))
    {
        errorText->Text = "Start failure: " + hr.ToString();
        return;
    }

    // Fill the array with sound data
    for (int index = 0, second = 0; second < 5; second++)
    {
        for (int cycle = 0; cycle < 441; cycle++)
        {
            for (int sample = 0; sample < 100; sample++)
            {
                short value = sample < 50 ? 32767 : -32768;
                soundData[index++] = value & 0xFF;
                soundData[index++] = (value >> 8) & 0xFF;
            }
        }
    }

    // Make the button visible
    submitButton->Visibility = Windows::UI::Xaml::Visibility::Visible;
}

void MainPage::OnSubmitButtonClick(Object^ sender, RoutedEventArgs^ args)
{
    // Create a buffer to reference the byte array
    XAUDIO2_BUFFER buffer = { 0 };
    buffer.AudioBytes = 2 * 5 * 44100;
    buffer.pAudioData = soundData;
    buffer.Flags = XAUDIO2_END_OF_STREAM;
    buffer.PlayBegin = 0;
    buffer.PlayLength = 5 * 44100;

    // Submit the buffer
    HRESULT hr = pSourceVoice->SubmitSourceBuffer(&buffer);

    if (FAILED(hr))
    {
        errorText->Text = "SubmitSourceBuffer failure: " + hr.ToString();
        submitButton->Visibility = Windows::UI::Xaml::Visibility::Collapsed;
        return;
    }
}
```

The `CreateSourceVoice` call requires a `WAVEFORMATEX` structure, which will be familiar to anyone who has worked with audio in the Win32 API. This structure defines the nature of the audio data you'll be using for this particular voice. (Different source voices can use different formats.) For PCM, only three numbers are really relevant: the sampling rate (44,100 in this example), the size of each sample (2 bytes or 16 bits) and the number of channels (1 here). The other fields are based on these: The `nBlockAlign` field is `nChannels` times `wBitsPerSample` divided by 8, and `nAvgBytesPerSec` field is the product of `nSamplesPerSec` and `nBlockAlign`. But in this example I've shown all the fields with explicit values.

Once the `IXAudio2SourceVoice` object is obtained, the `Start` method can be called on it. At this point, `IXAudio2` is conceptually playing, but we haven't actually given it any audio data to play. A `Stop` method is also available, and a real program would use these two methods to control when sounds should and shouldn't be playing.

All four of these calls are not as simple as they appear in this code! They all have additional arguments, but convenient defaults are defined and I've simply chosen to accept those defaults for now.

Virtually all function and method calls in DirectX return `HRESULT` values to indicate success or failure. There are different strategies for dealing with these errors. I've chosen simply to display the error code using the `TextBlock` defined in the XAML file, and stop further processing.

PCM Audio Data

The constructor concludes by filling up the `soundData` array with audio data, but the array isn't actually forked over to the `IXAudio2SourceVoice` until the button is pressed.

Sound is vibration, and humans are sensitive to vibrations in the air roughly in the range of 20 Hz (or cycles per second) to 20,000 Hz. Middle C on the piano is approximately 261.6 Hz.

Suppose you're working with a sampling rate of 44,100 Hz and 16-bit samples and you wish to generate audio data for a waveform at a frequency of 4,410 Hz, which is just beyond the highest key on a piano. Each cycle of such a waveform requires 10 samples of signed 16-bit values. These 10 values would be repeated 4,410 times for each second of sound.

A waveform at a frequency of 441 Hz—very close to 440 Hz, corresponding to the A above middle C used as a tuning

standard—is rendered with 100 samples. This cycle would be repeated 441 times for each second of sound.

Because PCM involves a constant sampling rate, low-frequency sounds seem to be sampled and rendered at a much higher resolution than high-frequency sounds. Isn't this a problem? Doesn't a 4,410 Hz waveform rendered with just 10 samples have a considerable amount of distortion compared with the 441 Hz waveform?

It turns out that any quantization distortion in PCM occurs at frequencies greater than half the sampling rate. (This is known as the Nyquist frequency after Bell Labs engineer Harry Nyquist.) One reason a sampling frequency of 44,100 Hz was chosen for CD audio is that the Nyquist frequency is 22,050 Hz, and human hearing maxes out at about 20,000 Hz. In other words, at a sampling rate of 44,100 Hz, the quantization distortion is inaudible to humans.

The `SimpleAudio` program generates an algorithmically simple waveform—a square wave at a frequency of 441 Hz. There are 100 samples per cycle. In each cycle the first 50 are maximum positive values (32,767 when dealing with short integers) and the next 50 are maximum negative values (-32,768). Notice that these short values must be stored in the byte array with the low byte first:

```
soundData[index + 0] = value & 0xFF;
soundData[index + 1] = (value >> 8) & 0xFF;
```

So far, nothing has actually played. This happens in the `Click` handler for the `Button`. The `XAUDIO2_BUFFER` structure is used to reference the byte array with a count of the bytes and a duration specified as the number of samples. This buffer is passed to the `SubmitSourceBuffer` method of the `IXAudio2SourceVoice` object. If the `Start` method has already been called (as it has in this example) then the sound begins playing immediately.

I suspect I don't have to mention that the sound plays asynchronously. The `SubmitSourceBuffer` call returns immediately while a separate thread is devoted to the actual process of shoveling data to the sound hardware. The `XAUDIO2_BUFFER` passed to `SubmitSourceBuffer` can be discarded after the call—as it is in this program when the `Click` handler is exited and the local variable goes out of scope—but the actual array of bytes must remain in accessible memory. Indeed, your program can manipulate these bytes as the sound is playing. However, there are much better techniques (involving callback methods) that let your program dynamically generate sound data.

Without using a callback to determine when the sound has completed, this program needs to retain the `soundData` array for the program's duration.

You can press the button multiple times, and each call effectively queues up another buffer to be played when the previous buffer finishes. If the program moves to the background, the sound is muted but it continues to play in silence. In other words, if you click the button and move the program to the background for at least 5 seconds, nothing will be playing when the program returns to the foreground.

The Characteristics of Sound

Much of the sound we hear in daily life comes simultaneously from a variety of different sources and, hence, is quite complex. However, in some cases—and particularly when dealing with musical sounds—individual tones can be defined with just a few characteristics:

- Amplitude, which is interpreted by our senses as volume.
- Frequency, which is interpreted as pitch.

Figure 4 Sound Data That Changes in Volume for SoundCharacteristics

```
for (int index = 0, sample = 0; sample < samples; sample++)
{
    double t = 1;

    if (sample < 2 * samples / 5)
        t = sample / (2.0 * samples / 5);

    else if (sample > 3 * samples / 5)
        t = (samples - sample) / (2.0 * samples / 5);

    double amplitude = pow(2, 15 * t) - 1;
    short waveform = sample % 100 < 50 ? 1 : -1;
    short value = short(amplitude * waveform);

    volumeSoundData[index++] = value;
    volumeSoundData[index++] = value;
}
```

Figure 5 Sound Data That Changes in Frequency for SoundCharacteristics

```
double angle = 0;

for (int index = 0, sample = 0; sample < samples; sample++)
{
    double t = 1;

    if (sample < 2 * samples / 5)
        t = sample / (2.0 * samples / 5);

    else if (sample > 3 * samples / 5)
        t = (samples - sample) / (2.0 * samples / 5);

    double frequency = 220 * pow(2, 2 * t);
    double angleIncrement = 360 * frequency / waveform.nSamplesPerSec;
    angle += angleIncrement;

    while (angle > 360)
        angle -= 360;

    short value = angle < 180 ? 32767 : -32767;
    pitchSoundData[index++] = value;
    pitchSoundData[index++] = value;
}
```

- Space, which can be mimicked in audio playback with multiple speakers.
- Timbre, which is related to the mix of overtones in a sound and represents the perceived difference between a trumpet and a piano, for example.

The SoundCharacteristics project demonstrates these four characteristics in isolation. It keeps the 44,100 sample rate and 16-bit samples of the SimpleAudio project but generates sound in stereo. For two channels of sound, the data must be interleaved: a 16-bit sample for the left channel, followed by a 16-bit sample for the right channel.

The MainPage.xaml.h header file for SoundCharacteristics defines some constants:

```
static const int sampleRate = 44100;
static const int seconds = 5;
static const int channels = 2;
static const int samples = seconds * sampleRate;
```

It also defines four arrays for sound data, but these are of type short rather than byte:

```
short volumeSoundData[samples * channels];
short pitchSoundData[samples * channels];
short spaceSoundData[samples * channels];
short timbreSoundData[samples * channels];
```

Using short arrays makes the initialization easier because the 16-bit waveform values don't need to be broken in half. A simple cast allows the array to be referenced by the XAUDIO2_BUFFER when submitting the sound data. These arrays have double the number of bytes as the array in SimpleAudio because I'm using stereo in this program.

All four of these arrays are initialized in the MainPage constructor. For the volume demonstration, a 441 Hz square wave is still involved, but it starts at zero volume, gets progressively louder over the first 2 seconds, and then declines in volume over the last 2 seconds. **Figure 4** shows the code to initialize volumeSoundData.

Human perception of volume is logarithmic: Each doubling of a waveform's amplitude is equivalent to a 6 dB increase in volume. (The 16-bit amplitude used for CD audio has a dynamic range of 96 decibels.) The code shown in **Figure 4** to alter the volume first calculates a value of t that increases linearly from 0 to 1, and then decreases back to 0. The amplitude variable is calculated using the

pow function and ranges from 0 to 32,767. This is multiplied by a square wave that has values of 1 and -1. The result is added to the array twice: first for the left channel, then for the right channel.

Human perception of frequency is also logarithmic. Much of the world's music organizes pitch around the interval of an octave, which is a doubling of frequency. The first two notes of the chorus of "Somewhere over the Rainbow" are an octave leap whether it's sung by a bass or a soprano. **Figure 5** shows code that varies pitch over the range of two octaves: from 220 to 880 based on a value of t that (like the volume example) goes from 0 to 1 and then back down to 0.

In earlier examples I chose a frequency of 441 Hz because it divides cleanly into the sampling rate of 44,100. In the general case, the sampling rate isn't an integral multiple of the frequency, and hence there can't be an integral number of samples per cycle. Instead, this program maintains a floating point angleIncrement variable that is proportional to the frequency and used to increase an angle value that ranges from 0 to 360. This angle value is then used to construct the waveform.

The demonstration for space moves the sound from the center to the left channel, then to the right, then back to the center. For the timbre demo, the waveform starts with a sine curve at 441 Hz. A sine curve is the mathematical representation of the most fundamental type of vibration—the solution of the differential equation where force is inversely proportional to displacement. All other periodic waveforms contain harmonics, which are also sine waves but with frequencies that are integral multiples of the base frequency. The timbre demo changes the waveform smoothly from a sine wave to a triangle wave, to a square wave, to a sawtooth wave, while increasing the harmonic content of the sound.

The Bigger Picture

Although I've just demonstrated how you can control volume, pitch, space, and timbre by generating sound data for a single IXAudio2SourceVoice object, the object itself includes methods to change the volume and space, and even the frequency. (A "3D" space facility is also supported.) Although it's possible to generate composite sound data that combines a bunch of individual tones with a single-source voice, you can create multiple IXAudio2SourceVoice objects and play them all together through the same mastering voice.

In addition, XAudio2 defines an IXAudioSubmixVoice that allows you to define filters and other effects, such as reverberation or echo. Filters have the ability to change the timbre of existing tones dynamically, which can contribute greatly to creating interesting and realistic musical sounds.

Perhaps the most essential enhancement beyond what I've shown in these two programs requires working with XAudio2 callback functions. Instead of allocating and initializing big chunks of sound data as these two programs do, it makes much more sense for a program to generate sound data dynamically as it's being played. ■

CHARLES PETZOLD is a longtime contributor to MSDN Magazine, and the author of "Programming Windows, 6th edition" (O'Reilly Media, 2012), a book about writing applications for Windows 8. His Web site is charlespetzold.com.

THANKS to the following technical expert for reviewing this article: Scott Selfon



Lowering Higher Education, Again

A year ago in this column, I told you how the Internet would hammer the higher education industry the same way it's hammered the newspaper industry (and the travel industry, and the music industry and ... you get the idea). I never imagined how quickly that hammering would start.

There's a new word—a geek acronym, naturally—for that hammer: MOOC (rhymes with fluke). It stands for Massive Open Online Course, and while I delight in the MOOC concept, I have wondered about its prospects. I wondered if reputable providers would produce enough high-quality content. Well, they've jumped onto that. Three large organizations dominate the market: Udacity, started by Sebastian Thrun from Stanford University (udacity.com); edX, a collaboration of Harvard University, Massachusetts Institute of Technology and the University of California, Berkeley (edx.org); and Coursera, an umbrella of 33 universities, including Brown, Penn, Duke, Stanford and Caltech (coursera.org). Online learning isn't limited to the for-profit University of Phoenix any more.

I wondered if the content would adapt to the online learning environment. Anant Agarwal at MIT had more than 150,000 students from all over the world sign up for his inaugural MOOC on circuit design and analysis. In addition to carefully produced videos, he provided lab exercises with an online circuit sandbox, containing virtual signal probes, sweep generators and oscilloscopes. I can see this simulator idea catching on, thereby allowing MOOCs to handle lab classes such as freshman chemistry.

I wondered how students could meet other students, and speculated that bars might sponsor it. The recreation department in my hometown of Ipswich, Mass., has since announced the formation of Ipswich University (ipswichu.org). Any time two or more people want to get together for a class, Ipswich University arranges a meeting place somewhere in town, with workspace and computing facilities. You will have to bring your own beer (so far, anyway).

My Harvard Extension students love the online format. Apart from the obvious advantages of slotting my class anywhere in their schedules, and not having to fight traffic into Cambridge and pay for parking, they like being able to pause a lecture so they can take notes or look something up. They also like replaying sections they didn't understand at first, or when they miss the punch line of a joke. I, on the other hand, find it somewhat depressing to have only one or two students attend live, instead of 100 like I used to. My teaching assistants have heard most of my jokes, often many

times, and don't laugh the way virgin students do. I'm considering adding a canned laugh track to my lectures.

Of course, as soon as online education starts producing useful results, governments start screwing it up. The state of Minnesota has banned Coursera from offering courses there because Coursera had not received authorization from the state to do so (bit.ly/WqZfQ8). Something is seriously wrong when students have to flee to neighboring Iowa to pursue a free online education. Perhaps we need a MOOC on the works of Ayn Rand.

The Hammer Will Fall

Online learning is currently being marketed as an enrichment activity: "lifelong learning" is the latest catchphrase. With the fast pace of technology-driven change in all fields, I certainly see the need for higher education to evolve from one huge bolus after high school to more of a steady lifelong drip. I spoke with a longtime reader named Ken, who's currently pursuing an online MBA, and he loves it. As a father of three, working a full-time job, he couldn't advance his education any other way.

But it won't be long before online challenges traditional universities for undergraduate degrees. Nascent MOOCs and online programs haven't yet made serious inroads, but the economics ensure that they're headed that way, throttle wide open and turbocharger howling. Coursera is even exploring college credit for its MOOCs (see bit.ly/SSZYMI).

The hammer will fall first on the Hampshire Colleges and Colgate Universities of the world. They don't have the reputation of a Harvard to attract full-fare parents when MOOCs cost so much less, nor the endowment of a Harvard to sustain themselves without it. I wonder which institutions I'll be writing deathwatches or obituaries for in this column next year.

I'll know that online learning has triumphed when the first person with an online degree gets accepted to medical school. Freshman medical students still need to dissect cadavers in person. For now, anyway. ■

DAVID S. PLATT teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.

Powerful Tools for Developers



Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents programmatically.
- Fast and lightweight 32 and 64-bit components for .Net, COM and WinRT applications.
- Create, fill-out and annotate PDF forms.



Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package.
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
- Includes our Microsoft certified PDF Converter printer driver.
- Export PDF documents into other formats such as JPeg, Word, Excel or Silverlight.



Advanced HTML to PDF & XAML

- Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver.
- Easy Integration and deployment within developer's applications.
- WebkitPDF is based on the Webkit Open Source library and Amyuni PDF Creator.



High Performance PDF Printer Driver



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2008 and Windows 8.
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language.
- Easy licensing and deployment to fit system administrator's requirements.



AMYUNI

All development tools available at

www.amyuni.com

USA and Canada

Toll Free: 1866 926 9864
Support: 514 868 9227
sales@amyuni.com

Europe

UK: 0800-015-4682
Germany: 0800-183-0923
France: 0800-911-248



Objective-C

Succinctly

by Ryan Hodson

